Counterpart Documentation

Release 1.4.0

Christopher Davis

August 01, 2014

Contents

1 Matchers **3**
- 1.1 Built in Matchers . . . 3

2 Logical Combinations **5**
- 2.1 Logical Not (Negating Matchers) . . . 5
- 2.2 Logical And . . . 5
- 2.3 Logical Or . . . 6
- 2.4 Logical Xor . . . 6

3 Custom Matchers **7**
- 3.1 Better Negation Messages . . . 8
- 3.2 Mismatch Descriptions . . . 8
- 3.3 Using Custom Matchers for Assertions . . . 9

4 Quickstart **11**
- 4.1 Matchers . . . 11
- 4.2 Logical Combinations . . . 11
- 4.3 Assertions . . . 12

Counterpart is a *matching* framework for PHP and is used to compare values in an object oriented way.

Some example use cases:

1. A testing framework could use Counterpart's matchers and `Assert` class for assertions.
2. A mock object library could use Counterpart to match argument expectations.
3. A validation library could use Counterpart to check that values match expectations.

Contents:

CHAPTER 1

Matchers

This document is a brief overview of all the matchers that counter part provides. For a more in depth look at the matchers, head over to the api documentation.

For information about custom matchers see *Custom Matchers*.

1.1 Built in Matchers

- `Anything`: Match literally anything.
- `Callback`: Run the actual value through a user defined callback. If the callback returns a truthy value it's a match.
- `Contains`: Check if an array of `Traversable` contains a value
- `Count`: Check if an array, `Traversable`, or `Countable` matches an expected count.
- `FileExists`: Check if a file exists.
- `GreaterThan`: Check if a value is greater than an expected number.
- `HasKey`: Check if an array or `ArrayAccess` contains a given key.
- `HasProperty`: Check if an object has a given property. This can be configured to only match public properties.
- `IsEmpty`: Check if a value is empty (eg. `empty($actual)`).
- `IsEqual`: Check if two values are equal – can be configured to use strict equality.
- `IsFalse`: Check if an actual value is exactly equal to `false`.
- `IsFalsy`: Check if an actual value is *falsy*. These are things like `"no"`, `0`, or `"0"`.
- `IsInstanceOf`: Check of an actual value is an instance of a given class or interfact.
- `IsJson`: Check if an actual value is a valid *JSON <http://www.json.org/>* string.
- `IsNull`: Check if an actual value is exactly equal to `null`.
- `IsTrue`: Check if an actual value is exactly equal to `true`.
- `IsTruthy`: Check if an actual value is *truthy*. These are things like `"yes"`, `1`, or `"1"`.
- `IsType`: Check if an object is an internal type.
- `LessThan`: Check if an actual value is less than a given number.

- `LogicalAnd`: Combine one or more matchers with a conjuction. See *Logical Combinations*. Will match if all sub-matchers match.
- `LogicalNot`: Negate a matcher. See *Logical Combinations*.
- `LogicalOr`: Combine one or more matchers with a disjunction. See *Logical Combinations*. Will match if an only if one of its sub-matchers matches.
- `LogicalXor`: Combine one or more matchers with an XOR. `LogicalXor` will match if an only if exactly one of its sub-matchers matches.
- `MatchesRegex`: Checks a string (or object with a `__toString` method) against a regular expression.
- `PhptFormat`: Checks a string against a *phpt format <http://qa.php.net/phpt_details.php#expectf_section>*.
- `StringContains`: Checks to see if a string contains an expected value.

CHAPTER 2

Logical Combinations

Counterpart provides a set of matchers that allow users to create logical combinations of one or more matchers.

2.1 Logical Not (Negating Matchers)

A matcher can be negated with `LogicalNot`.

```
<?php
use Counterpart\Matchers;

$matcher = Matchers::logicNot(Matchers::hasKey('a_key'));
$matcher->matches(['a_key' => '']); // false
echo $matcher; // "is an array or ArrayAccess without the key a_key"
```

There are a fair amount negative matcher factories already set up. The above could be more simply written.

```
<?php
use Counterpart\Matchers;

$matcher = Matchers::doesNotHaveKey('a_key'));
$matcher->matches(['a_key' => '']); // false
echo $matcher; // "is an array or ArrayAccess without the key a_key"
```

2.2 Logical And

`LogicalAnd` can be used to combine one or more matchers with an `AND` or conjuction. When all sub-matchers match a value, `LogicalAnd` will return `true`. Checking to see if a value is in a range is a great example of this.

```
<?php
use Counterpart\Matchers;

$matcher = Matchers::logicalAnd(
    Matchers::greaterThan(10),
    Matchers::lessThan(100)
);
$matcher->matches(11); // true
$matchers->matches(101); // false
```

2.3 Logical Or

`LogicalOr` can be used to combine one or more matchers with an `OR` or disjunction. If at least one sub-matcher matches the value, `LogicalOr` will also match. Checking that a value is greater than or equal to another is a great example of this.

```
<?php
use Counterpart\Matchers;

// same as Matchers::greaterThanOrEqual(10);
$matcher = Matchers::logicalOr(
    Matchers::equalTo(10),
    Matchers::greaterThan(10)
);
$matcher->matches(10); // true
$matcher->matches(20); // true
$matcher->matches(9); // false
```

2.4 Logical Xor

`LogicalXor` can be used to combine one or more matchers with an `XOR`. `LogicalXor` will return true if one and only one of the sub-matchers matches. The above greater than or equal to example could be written using `logicalXor`.

```
<?php
use Counterpart\Matchers;

$matcher = Matchers::logicalXor(
    Matchers::equalTo(10),
    Matchers::greaterThan(10)
);
$matcher->matches(10); // true
$matcher->matches(20); // true
$matcher->matches(9); // false
```

CHAPTER 3

Custom Matchers

One of the goals of Counterpart is to make it easy to create custom matchers. The `Counterpart\Matcher` interface only contains two methods: `matches` and `__toString`.

Let's make a custom matcher that checks to see if a value is in a range.

```php
<?php
namespace Acme\CounterpartExample;

use Counterpart\Matcher;

/**
 * Matches a value if its between $min and $max
 */
class RangeMatcher implements Matcher
{
    private $min;
    private $max;

    public function __construct($min, $max)
    {
        $this->min = $min;
        $this->max = $max;
    }

    /**
     * Matches checks an actual value against the expectations.
     */
    public function matches($actual)
    {
        return $actual > $this->min && $actual < $this->max;
    }

    /**
     * This should return a textual description of the what the matcher
     * is trying to accomplish.
     */
    public function __toString()
    {
        return sprintf('is a value between %d and %d', $this->min, $this->max);
    }
}
```

3.1 Better Negation Messages

By default Counterpart's *LogicalNot* will replace the starting *is* in a matchers description with *is not*. That's not always so great for generating a negation error message.

For a more customized negative message, a matcher can implement `Counterpart\Negative`.

```
<?php
namespace Acme\CounterpartExample;

use Counterpart\Matcher;
use Counterpart\Negative;

/**
 * Matches a value if its between $min and $max
 */
class RangeMatcher implements Matcher, Negative
{
    // all the stuff above

    /**
     * 'LogicalNot' will call this this to generate a nice negative message.
     */
    public function negativeMessage()
    {
        // this is what Counterpart would have done anyway
        return sprintf('is not a value between %d and %d', $this->min, $this->max);
    }
}
```

3.2 Mismatch Descriptions

When Counterpart does assertions it will call a matchers `__toString` method as part of the error description. Sometimes this isn't enough – sometimes it doesn't provide enough context for the user or developer to take action.

Custom matchers may implement `Counterpart\Describer` to generate a more thorough description of a mismatch.

```
<?php
namespace Acme\CounterpartExample;

use Counterpart\Matcher;
use Counterpart\Negative;
use Counterpart\Describer;

/**
 * Matches a value if its between $min and $max
 */
class RangeMatcher implements Matcher, Negative, Describer
{
    // all the stuff above

    /**
     * 'Counterpart\Assert::assertThat' will call this method to to generate
     * a more thorough error description.
     */
```

```
    public function describeMismatch($actual)
    {
        if ($actual < $this->min) {
            return 'the value was below the minimum';
        }

        if ($actual > $this->max) {
            return 'the value was above the maximum';
        }

        // the method doesn't know what to do, so decline to do anything.
        return Describer::DECLINE_DESCRIPTION;
    }
}
```

3.3 Using Custom Matchers for Assertions

Simply pass an instance of the custom matcher as the first argument to `Counterpart\Assert::assertThat`.

```
<?php
use Counterpart\Assert;
use Acme\CounterpartExample\RangeMatcher;

$actualValue = 9;
Assert::assertThat(new RangeMatcher(1, 10), $actualValue);
```

CHAPTER 4

Quickstart

Counterpart can be installed via composer, just add it it to your `composer.json`.

```
{
    "name": "somevendor/someproject",
    "require": {
        "counterpart/counterpart": "~1.4"
    }
}
```

Then simply `composer install` or `composer update`.

Counterpart provides two traits full of static factory and helper methods.

- `Counterpart\Matchers`
- `Counterpart\Assert`

4.1 Matchers

The `Counterpart\Matchers` trait comes with a set of static factory methods to make using matchers easy.

```
<?php

use Counterpart\Matchers;

$matcher = Matchers::hasKey('a_key');
$matcher->matches(['a_key' => '']); // true
echo $matcher; // "is an array or ArrayAccess with the key a_key"
```

Every matcher object implements `Counterpart\Matcher` whose `match` method does all the heavy lifting. The matcher interface also includes a `__toString` method which will return a textual description of what's being looked for.

4.2 Logical Combinations

Counterpart provides a set of matchers that allow users to create logical combinations of one or more matchers.

See *Logical Combinations* for more.

4.3 Assertions

The `Counterpart\Assert` trait provides assertions: matchers wrapped up in a helper that throws a `Counterpart\Exception\AssertionFailed` exception when the matcher fails.

```
<?php
use Counterpart\Assert;

Assert::assertEquals(10, 10, "two values that are equal are not matching as equal, something is wrong
Assert::assertFileExists(__FILE__);
```

It's also passible to use a custom matcher with the `Assert` trait directly. Simple paces an instance of `Counterpart\Matcher` as the first argument to `Assert::assertThat`.

```
<?php
use Counterpart\Assert;
use Counterpart\Matcher\IsEqual;

Assert::assertThat(new IsEqual(1), 1, "1 != 1, something is very broken");
```