Imagine Documentation

Release 1.0.1

Bulat Shakirzyanov

Sep 27, 2018

Contents

1 Contribute: **3**

2 Ask a question: **5**

3 Usage: **7**

3.1 Quick introduction . . . 7
3.1.1 Installation . . . 7
3.1.2 Basic usage . . . 8
3.1.3 Advanced Examples . . . 12
3.1.4 Architecture . . . 14
3.2 Metadata . . . 14
3.2.1 Access image metadata . . . 14
3.2.2 Change the metadata reader . . . 15
3.2.3 Create your own metadata reader . . . 17
3.3 Imagine's coordinates system . . . 18
3.3.1 Classes . . . 18
3.3.2 PointInterface . . . 18
3.3.3 BoxInterface . . . 19
3.4 Drawing shapes on an image . . . 19
3.4.1 Example . . . 19
3.4.2 Text . . . 20
3.5 Colors . . . 20
3.5.1 Palette Class . . . 20
3.5.2 About colorspaces support . . . 21
3.5.3 Color Class . . . 21
3.5.4 Profile Class . . . 22
3.6 Layers manipulation . . . 22
3.6.1 Disclaimer . . . 22
3.6.2 Layers Manipulation . . . 22
3.6.3 Generate Animated gif . . . 23
3.6.4 Animated gif frame manipulation . . . 24
3.7 Image effects . . . 25
3.7.1 Example . . . 25
3.7.2 Effects API . . . 25
3.8 Image filters and delayed processing . . . 27
3.8.1 Image Transformations, aka Lazy Processing . . . 27
3.8.2 Filter Application Order . . . 28

3.8.3 Filters . . . 28
3.9 Handling exceptions . . . 28
3.9.1 Exception interface . . . 29
3.9.2 Exception classes . . . 29

4 Api docs: **31**

5 A couple of words in defense **33**

6 Indices and tables **35**

Imagine is a OOP library for image manipulation built in PHP 5.3 using the latest best practices and thoughtful design that should allow for decoupled and unit-testable code.

```php
<?php

$imagine = new Imagine\Gd\Imagine();
// or
$imagine = new Imagine\Imagick\Imagine();
// or
$imagine = new Imagine\Gmagick\Imagine();

$size    = new Imagine\Image\Box(40, 40);

$mode    = Imagine\Image\ImageInterface::THUMBNAIL_INSET;
// or
$mode    = Imagine\Image\ImageInterface::THUMBNAIL_OUTBOUND;

$imagine->open('/path/to/large_image.jpg')
    ->thumbnail($size, $mode)
    ->save('/path/to/thumbnail.png')
;
```

Enjoy!

CHAPTER 1

Contribute:

Your contributions are more than welcome !

Start by forking Imagine repository, write your feature, fix bugs, and send a pull request.

If you're a beginner, you will find some guidelines about code contributions at Symfony

CHAPTER 2

Ask a question:

We're on IRC: `#php-imagine` on Freenode

CHAPTER 3

Usage:

3.1 Quick introduction

ImagineInterface (`Imagine\Image\ImagineInterface`) and its implementations is the main entry point into Imagine. You may think of it as a factory for `Imagine\Image\ImageInterface` as it is responsible for creating and opening instances of it and also for instantiating `Imagine\Image\FontInterface` object.

The main piece of image processing functionality is concentrated in the `ImageInterface` implementations (one per driver - e.g. `Imagick\Image`)

The main idea of Imagine is to avoid driver specific methods spill outside of this class and couple of other internal interfaces (`Draw\DrawerInterface`), so that the filters and any other image manipulations can operate on `ImageInterface` through its public API.

3.1.1 Installation

The recommended way to install Imagine is through Composer. Composer is a dependency management library for PHP.

Here is an example of composer project configuration that requires imagine version 0.5.

```
{
    "require": {
        "imagine/imagine": "~0.5.0"
    }
}
```

Install the dependencies using composer.phar and use Imagine :

```
php composer.phar install
```

```php
<?php
require 'vendor/autoload.php';

$imagine = new Imagine\Gd\Imagine();
```

3.1.2 Basic usage

Open Existing Images

To open an existing image, all you need is to instantiate an image factory and invoke `ImagineInterface::open()` with `$path` to image as the argument

```php
<?php

$imagine = new Imagine\Gd\Imagine();
// or
$imagine = new Imagine\Imagick\Imagine();

$image = $imagine->open('/path/to/image.jpg');
```

Tip: Read more about ImagineInterface

The `ImagineInterface::open()` method may throw one of the following exceptions:

- `Imagine\Exception\InvalidArgumentException`
- `Imagine\Exception\RuntimeException`

Tip: Read more about exceptions

Now that you've opened an image, you can perform manipulations on it:

```php
<?php

use Imagine\Image\Box;
use Imagine\Image\Point;

$image->resize(new Box(15, 25))
   ->rotate(45)
   ->crop(new Point(0, 0), new Box(45, 45))
   ->save('/path/to/new/image.jpg');
```

Tip: Read more about ImageInterface Read more about coordinates

Resize Images

Resize an image is very easy, just pass the box size you want as argument :

```php
<?php

use Imagine\Image\Box;
use Imagine\Image\Point;

$image->resize(new Box(15, 25))
```

You can also specify the filter you want as second argument :

```php
<?php

use Imagine\Image\Box;
use Imagine\Image\Point;
use Imagine\Image\ImageInterface;

// resize with lanczos filter
$image->resize(new Box(15, 25), ImageInterface::FILTER_LANCZOS);
```

Available filters are `ImageInterface::FILTER_*` constants.

Note: GD only supports `ImageInterface::RESIZE_UNDEFINED` filter.

Create New Images

Imagine also lets you create new, empty images. The following example creates an empty image of width 400px and height 300px:

```php
<?php

$size  = new Imagine\Image\Box(400, 300);
$image = $imagine->create($size);
```

You can optionally specify the fill color for the new image, which defaults to opaque white. The following example creates a new image with a fully-transparent black background:

```php
<?php

$palette = new Imagine\Image\Palette\RGB();
$size  = new Imagine\Image\Box(400, 300);
$color = $palette->color('#000', 0);
$image = $imagine->create($size, $color);
```

To use a solid background color, for example orange, provide an alpha of 100.

```php
<?php

$palette = new Imagine\Image\Palette\RGB();
$size  = new Imagine\Image\Box(400, 300);
$color = $palette->color('#ff9900', 100);
$image = $imagine->create($size, $color);
```

Save Images

Images are saved given a path and optionally options.

The following example opens a Jpg image and saves it as Png format :

```
<?php

$imagine = new Imagine\Imagick\Imagine();

$imagine->open('/path/to/image.jpg')
   ->save('/path/to/image.png');
```

Three options groups are currently supported : quality, resolution and flatten.

Tip: Default values are 75 for Jpeg quality, 7 for Png compression level and 72 dpi for x/y-resolution.

Note: GD does not support resolution options group

The following example demonstrates the basic quality settings.

```
<?php

$imagine = new Imagine\Imagick\Imagine();

$imagine->open('/path/to/image.jpg')
   ->save('/path/to/image.jpg', array('jpeg_quality' => 50)) // from 0 to 100
   ->save('/path/to/image.png', array('png_compression_level' => 9)); // from 0 to 9
```

The following example opens a Jpg image and saves it with it with 150 dpi horizontal resolution and 120 dpi vertical resolution.

```
<?php

use Imagine\Image\ImageInterface;

$imagine = new Imagine\Imagick\Imagine();

$options = array(
    'resolution-units' => ImageInterface::RESOLUTION_PIXELSPERINCH,
    'resolution-x' => 150,
    'resolution-y' => 120,
    'resampling-filter' => ImageInterface::FILTER_LANCZOS,
);

$imagine->open('/path/to/image.jpg')->save('/path/to/image.jpg', $options);
```

Note: You **MUST** provide a unit system when setting resolution values. There are two available unit systems for resolution : `ImageInterface::RESOLUTION_PIXELSPERINCH` and `ImageInterface::RESOLUTION_PIXELSPERCENTIMETER`.

The flatten option is used when dealing with multi-layers images (see the layers section for information). Image are saved flatten by default, you can avoid this by explicitly set this option to `false` when saving :

```
<?php
```

(continues on next page)

(continued from previous page)

```
use Imagine\Image\Box;
use Imagine\Image\ImageInterface;
use Imagine\Imagick\Imagine;

$imagine = new Imagine();

$imagine->open('/path/to/animated.gif')
        ->resize(new Box(320, 240))
        ->save('/path/to/animated-resized.gif', array('flatten' => false));
```

Tip: You **SHOULD NOT** flatten image only for animated gif and png images.

Of course, you can combine options :

```
<?php

use Imagine\Image\ImageInterface;

$imagine = new Imagine\Imagick\Imagine();

$options = array(
    'resolution-units' => ImageInterface::RESOLUTION_PIXELSPERINCH,
    'resolution-x' => 300,
    'resolution-y' => 300,
    'jpeg_quality' => 100,
);

$imagine->open('/path/to/image.jpg')->save('/path/to/image.jpg', $options);
```

Show Images

Images are shown (i.e. outputs the image content) given a format and optionally options.

The following example shows a Jpg image:

```
<?php

$imagine = new Imagine\Imagick\Imagine();

$imagine->open('/path/to/image.jpg')
   ->show('jpg');
```

Note: This will send a “Content-type” header.

It supports the same options groups as for the save method.

For example:

```
<?php

$imagine = new Imagine\Imagick\Imagine();
```

(continues on next page)

(continued from previous page)

```
$options = array(
    'resolution-units' => ImageInterface::RESOLUTION_PIXELSPERINCH,
    'resolution-x' => 300,
    'resolution-y' => 300,
    'jpeg_quality' => 100,
);

$imagine->open('/path/to/image.jpg')
    ->show('jpg', $options);
```

3.1.3 Advanced Examples

Image Watermarking

Here is a simple way to add a watermark to an image :

```
<?php

$watermark = $imagine->open('/my/watermark.png');
$image     = $imagine->open('/path/to/image.jpg');
$size      = $image->getSize();
$wSize     = $watermark->getSize();

$bottomRight = new Imagine\Image\Point($size->getWidth() - $wSize->getWidth(), $size->
↪getHeight() - $wSize->getHeight());

$image->paste($watermark, $bottomRight);
```

An Image Collage

Assume we were given the not-so-easy task of creating a four-by-four collage of 16 student portraits for a school yearbook. Each photo is 30x40 px and we need four rows and columns in our collage, so the final product will be 120x160 px.

Here is how we would approach this problem with Imagine.

```
<?php

use Imagine;

// make an empty image (canvas) 120x160px
$collage = $imagine->create(new Imagine\Image\Box(120, 160));

// starting coordinates (in pixels) for inserting the first image
$x = 0;
$y = 0;

foreach (glob('/path/to/people/photos/*.jpg') as $path) {
    // open photo
    $photo = $imagine->open($path);

    // paste photo at current position
    $collage->paste($photo, new Imagine\Image\Point($x, $y));
```

(continues on next page)

(continued from previous page)

```
    // move position by 30px to the right
    $x += 30;

    if ($x >= 120) {
        // we reached the right border of our collage, so advance to the
        // next row and reset our column to the left.
        $y += 40;
        $x = 0;
    }

    if ($y >= 160) {
        break; // done
    }
}

$collage->save('/path/to/collage.jpg');
```

Image Reflection Filter

```
<?php

class ReflectionFilter implements Imagine\Filter\FilterInterface
{
    private $imagine;

    public function __construct(Imagine\Image\ImagineInterface $imagine)
    {
        $this->imagine = $imagine;
    }

    public function apply(Imagine\Image\ImageInterface $image)
    {
        $size       = $image->getSize();
        $canvas     = new Imagine\Image\Box($size->getWidth(), $size->getHeight() *␣
↪2);
        $reflection = $image->copy()
            ->flipVertically()
            ->applyMask($this->getTransparencyMask($image->palette(), $size))
        ;

        return $this->imagine->create($canvas, $image->palette()->color('fff', 100))
            ->paste($image, new Imagine\Image\Point(0, 0))
            ->paste($reflection, new Imagine\Image\Point(0, $size->getHeight()));
    }

    private function getTransparencyMask(Imagine\Image\Palette\PaletteInterface
↪$palette, Imagine\Image\BoxInterface $size)
    {
        $white = $palette->color('fff');
        $fill  = new Imagine\Image\Fill\Gradient\Vertical(
            $size->getHeight(),
            $white->darken(127),
            $white
        );
```

(continues on next page)

(continued from previous page)

```
        return $this->imagine->create($size)
            ->fill($fill)
        ;
    }
}

$imagine = new Imagine\Gd\Imagine();
$filter  = new ReflectionFilter($imagine);

$filter->apply($imagine->open('/path/to/image/to/reflect.png'))
   ->save('/path/to/processed/image.png')
;
```

Tip: For step by step explanation of the above code see Reflection section of Introduction to Imagine

3.1.4 Architecture

The architecture is very flexible, as the filters don't need any processing logic other than calculating the variables based on some settings and invoking the corresponding method, or sequence of methods, on the `ImageInterface` implementation.

The `Transformation` object is an example of a composite filter, representing a stack or queue of filters, that get applied to an Image upon application of the `Transformation` itself.

Tip: For more information about `Transformation` filter see Transformation section of Introduction to Imagine

3.2 Metadata

Imagine 0.6 comes with an abstraction to read Image metadata.

3.2.1 Access image metadata

Metadata are read with a `Imagine\Image\Metadata\MetadataReaderInterface` and accessible through `Imagine\Image\ImageInterface::metadata` method that returns a `Imagine\Image\Metadata\MetadataBag` object:

```
<?php

use Imagine\Image\Metadata\ExifMetadataReader;

$image = $imagine->open('/path/to/image.jpg');
$metadata = $image->metadata();

// prints '/path/to/image.jpg'
print($metadata['filename']);
```

3.2.2 Change the metadata reader

Imagine comes bundled with two metadata readers: `Imagine\Image\Metadata\DefaultMetadataReader` and `Imagine\Image\Metadata\ExifMetadataReader`.

Imagine uses the default metadata reader by default. You can easily switch to the one you want using `Imagine\Image\ImagineInterface::setMetadataReader` method.

```
<?php

use Imagine\Image\Metadata\ExifMetadataReader;

$imagine->setMetadataReader(new ExifMetadataReader());
```

Default Metadata Reader

The default metadata reader is a basic reader that stores original information about the resource.

```
<?php

use Imagine\Image\Metadata\DefaultMetadataReader;

$image = $imagine
    ->setMetadataReader(new DefaultMetadataReader())
    ->open('chenille.jpg');
$metadata = $image->metadata();

var_dump($metadata->toArray());
```

The previous code might produce such output:

```
array(2) {
  'filepath' =>
  string(60) "/Users/romainneutron/Documents/workspace/Imagine/chenille.jpg"
  'uri' =>
  string(12) "chenille.jpg"
}
```

Exif Metadata Reader

Exif Metadata Reader gives the same base information as the default metadata reader and adds exif data provided by the Exif extension.

Note: Using the exif metadata reader adds a significant overhead to image processing.

```
<?php

use Imagine\Image\Metadata\ExifMetadataReader;

$image = $imagine
    ->setMetadataReader(new ExifMetadataReader())
    ->open('chenille.jpg');
$metadata = $image->metadata();
```

(continues on next page)

(continued from previous page)

```
var_dump($metadata->toArray());
```

The previous code should produce this output:

```
array(37) {
  'filepath' =>
  string(60) "/Users/romainneutron/Documents/workspace/Imagine/chenille.jpg"
  'uri' =>
  string(12) "chenille.jpg"
  'exif.ExposureTime' =>
  string(5) "1/120"
  'exif.FNumber' =>
  string(4) "11/5"
  'exif.ExposureProgram' =>
  int(2)
  'exif.ISOSpeedRatings' =>
  int(40)
  'exif.ExifVersion' =>
  string(4) "0221"
  'exif.DateTimeOriginal' =>
  string(19) "2014:04:06 16:11:59"
  'exif.DateTimeDigitized' =>
  string(19) "2014:04:06 16:11:59"
  'exif.ComponentsConfiguration' =>
  string(4) "\000"
  'exif.ShutterSpeedValue' =>
  string(9) "9488/1373"
  'exif.ApertureValue' =>
  string(9) "7801/3429"
  'exif.BrightnessValue' =>
  string(8) "4457/710"
  'exif.MeteringMode' =>
  int(3)
  'exif.Flash' =>
  int(16)
  'exif.FocalLength' =>
  string(6) "103/25"
  'exif.ColorSpace' =>
  int(1)
  'exif.ExifImageWidth' =>
  int(2048)
  'exif.ExifImageLength' =>
  int(1536)
  'exif.SensingMethod' =>
  int(2)
  'exif.ExposureMode' =>
  int(0)
  'exif.WhiteBalance' =>
  int(0)
  'exif.FocalLengthIn35mmFilm' =>
  int(30)
  'exif.SceneCaptureType' =>
  int(0)
  'exif.UndefinedTag:0xA433' =>
  string(5) "Apple"
  'exif.UndefinedTag:0xA434' =>
```

(continues on next page)

(continued from previous page)

```
  string(34) "iPhone 5s back camera 4.12mm f/2.2"
  'ifd0.Make' =>
  string(5) "Apple"
  'ifd0.Model' =>
  string(9) "iPhone 5s"
  'ifd0.XResolution' =>
  string(4) "72/1"
  'ifd0.YResolution' =>
  string(4) "72/1"
  'ifd0.ResolutionUnit' =>
  int(2)
  'ifd0.Software' =>
  string(5) "7.0.4"
  'ifd0.DateTime' =>
  string(19) "2014:04:06 16:11:59"
  'ifd0.YCbCrPositioning' =>
  int(1)
  'ifd0.Exif_IFD_Pointer' =>
  int(192)
  'ifd0.GPS_IFD_Pointer' =>
  int(1486)
}
```

3.2.3 Create your own metadata reader

Any metadata reader must implement `Imagine\Image\Metadata\MetadataReaderInterface`. However it's easier to extend `Imagine\Image\Metadata\AbstractMetadataReader` to avoid missing things and focus on the purpose of the reader.

Here's an example of a metadata reader that retrieves posix access information from a file:

```php
<?php

use Imagine\Image\Metadata\AbstractMetadataReader;

class PosixMetadataReader extends AbstractMetadataReader
{
    /**
     * {@inheritdoc}
     */
    protected function extractFromFile($file)
    {
        // if file is not local, forget it
        if (!stream_is_local($file)) {
            return array();
        }

        return array(
            'access' => posix_access($file),
        );
    }

    /**
     * {@inheritdoc}
     */
```

(continues on next page)

(continued from previous page)

```
    protected function extractFromData($data)
    {
        // posix informations about raw data in non-sense
        return array();
    }

    /**
     * {@inheritdoc}
     */
    protected function extractFromStream($resource)
    {
        if (!stream_is_local($file)) {
            return array();
        }

        if (false !== $data = @stream_get_meta_data($resource)) {
            return array(
                'access' => posix_access($data['uri']),
            );
        }

        return array();
    }
}
```

3.3 Imagine's coordinates system

The coordinate system use by Imagine is very similar to Cartesian Coordinate System, with some exceptions:

- Coordinate system starts at x,y (0,0), which is the top left corner and extends to right and bottom accordingly
- There are no negative coordinates, a point must always be bound to the box its located at, hence 0,0 and greater
- Coordinates of the point are relative its parent bounding box

3.3.1 Classes

The whole coordinate system is represented in a handful of classes, but most importantly - its interfaces:

- `Imagine\Image\PointInterface` - represents a single point in a bounding box
- `Imagine\Image\BoxInterface` - represents dimensions (width, height)

3.3.2 PointInterface

Every coordinate contains the following methods:

- `->getX()` - returns horizontal position of the coordinate
- `->getY()` - returns vertical position of a coordinate
- `->in(BoxInterface $box)` - returns `true` if current coordinate appears to be inside of a given bounding `$box`
- `->__toString()` - returns string representation of the current `PointInterface`, e.g. `(0, 0)`

Center coordinate

It is very well known use case when a coordinate is supposed to represent a center of something.

As part of showing off OO approach to image processing, I added a simple implementation of the core `Imagine\Image\PointInterface`, which can be found at `Imagine\Image\Point\Center`. The way it works is simple, it expects and instance of `Imagine\Image\BoxInterface` in its constructor and calculates the center position based on that.

```
<?php

$size = new Imagine\Image\Box(50, 50);

$center = new Imagine\Image\Point\Center($size);

var_dump(array(
    'x' => $center->getX(),
    'y' => $center->getY(),
));

// would output position of (x,y) 25,25
```

3.3.3 BoxInterface

Every box or image or shape has a size, size has the following methods:

- `->getWidth()` - returns integer width
- `->getHeight()` - returns integer height
- `->scale($ratio)` - returns a new `BoxInterface` instance with each side multiplied by `$ratio`
- `->increase($size)` - returns a new `BoxInterface`, with given `$size` added to each side
- `->contains(BoxInterface $box, PointInterface $start = null)` - checks that the given `$box` is contained inside the current `BoxInterface` at `$start` position. If no `$start` position is given, its assumed to be (0,0)
- `->square()` - returns integer square of current `BoxInterface`, useful for determining total number of pixels in a box for example
- `->__toString()` - returns string representation of the current `BoxInterface`, e.g. `100x100 px`
- `->widen($width)` - resizes box to given width, constraining proportions and returns the new box
- `->heighten($height)` - resizes box to given height, constraining proportions and returns the new box

3.4 Drawing shapes on an image

Imagine also provides a fully-featured drawing API, inspired by Python's PIL. To use the api, you need to get a drawer instance from you current image instance, using `ImageInterface::draw()` method.

3.4.1 Example

```php
<?php
$palette = new Imagine\Image\Palette\RGB();

$image = $imagine->create(new Box(400, 300), $palette->color('#000'));

$image->draw()
    ->ellipse(new Point(200, 150), new Box(300, 225), $image->palette()->color('fff
↪'));

$image->save('/path/to/ellipse.png');
```

The above example would draw an ellipse on a black 400x300px image, of white color. It would place the ellipse in the center of the image, and set its larger radius to 300px, with a smaller radius of 225px. You could also make the ellipse filled, by passing *true* as the last parameter

3.4.2 Text

As you've noticed from `DrawerInterface::text()`, there is also `Font` class. This class is a simple value object, representing the font. To construct a font, you have to pass the `$file` string (path to font file), `$size` value (integer value, representing size points) and `$color` (`Imagine\Image\Palette\Color\ColorInterface` instance). After you have a font instance, you can use one of its three methods to inspect any of the values it's been constructed with:

- `->getFile()` - returns font file path
- `->getSize()` - returns integer size in points (e.g. 10pt = 10)
- `->getColor()` - returns `Imagine\Image\Palette\Color\ColorInterface` instance, representing current font color
- `->box($string, $angle = 0)` - returns `Imagine\Image\BoxInterface` instance, representing the estimated size of the `$string` at the given `$angle` on the image

3.5 Colors

Imagine provides a fully-featured colors API with the Palette object :

3.5.1 Palette Class

Every image in Imagine is attached to a Palette. The palette handles the colors. Imagine provides two palettes :

```php
<?php

$palette = new Imagine\Image\Palette\RGB();
// or
$palette = new Imagine\Image\Palette\CMYK();
```

When creating a new Image, the default RGB palette is used. It can be easily customized. For example the following code creates a new Image object with a white background and a CMYK palette.

```php
<?php

$palette = new Imagine\Image\Palette\CMYK();
$imagine->create(new Imagine\Image\Box(10, 10), $palette->color('#FFFFFF'));
```

You can switch your palette at any moment, for example to turn a CMYK image in RGB mode :

```
<?php

$image = $imagine->open('my-cmyk-jpg.jpg');
$image->usePalette(new Imagine\Image\Palette\RGB())
      ->save('my-rgb-jpg.jpg');
```

Note: Switching to a palette is the same a changing the colorspace.

3.5.2 About colorspaces support

Drivers do not handle colorspace the same way. Whereas GD only supports RGB images, Imagick supports CMYK, RGB and Grayscale colorspaces. Gmagick only supports CMYK and RGB colorspaces.

3.5.3 Color Class

Color is a class in Imagine, and is created through a palette with two arguments in its constructor: the RGB color code and a transparency percentage. The following examples are equivalent ways of defining a fully-transparent white color.

```
<?php

$white = $palette->color('fff', 100);
$white = $palette->color('ffffff', 100);
$white = $palette->color('#fff', 100);
$white = $palette->color('#ffffff', 100);
$white = $palette->color(0xFFFFFF, 100);
$white = $palette->color(array(255, 255, 255), 100);
```

Note: CMYK colors does not support alpha parameters.

After you have instantiated an RGB color, you can easily get its Red, Green, Blue and Alpha (transparency) values:

```
<?php

var_dump(array(
   'R' => $white->getRed(),
   'G' => $white->getGreen(),
   'B' => $white->getBlue(),
   'A' => $white->getAlpha()
));
```

The same behavior is available for CMYK colors :

```
<?php

var_dump(array(
   'C' => $white->getCyan(),
   'M' => $white->getMagenta(),
   'Y' => $white->getYellow(),
```

(continues on next page)

(continued from previous page)

```
    'K' => $white->getKeyline()
));
```

3.5.4 Profile Class

You can apply ICC profile on any Image class with the `profile` method :

```
<?php

$profile = Imagine\Image\Profile::fromPath('your-ICC-profile.icc');
$image->profile($profile)
      ->save('my-rgb-jpg-profiled.jpg');
```

3.6 Layers manipulation

`ImageInterface` provides an access for multi-layers image such as PSD files or animated gif. By calling the `layers()` method, you will get an iterable layer collection implementing the `LayersInterface`. As you will see, a layer implements `ImageInterface`

3.6.1 Disclaimer

Imagine is a fluent API to use Imagick, Gmagick or GD driver. These drivers do not handle all multi-layers formats equally. For example :

- PSD format should be flatten before being saved. (libraries would split it into different files),
- animated gif must not be flatten otherwise the animation would be lost.
- Tiff files should be split in multiple files or the result might be a pile of HD and thumbnail
- GD does not support layers.

You have to run tests against the formats you are using and their support by the driver you want before deploying in production.

3.6.2 Layers Manipulation

Imagine `LayersInterface` implements PHP's ArrayAccess, IteratorAggregate and Countable interfaces.

This provides many ways to manipulate layers.

Count Layers

```
<?php
$image = $imagine->open('image.jpg');

echo "Image contains " . count($image->layers()) . " layers";
```

Layers Iterations

```
<?php
$image = $imagine->open('image.jpg');

foreach ($image->layers() as $layer) {
    // ...
}
```

Layers Manipulation

Imagine provides an object oriented interface to manipulate layers :

```
<?php
$image = $imagine->open('image.jpg');
$layers = $image->layers();

$layers->get(0)->save('layer-0.jpg');          // access a layer
$layers->set(0, $imagine->open('image2.jpg')); // set layer at offset 0
$layers->add($imagine->open('image3.jpg'));     // push a new layer in layers
$layers->remove(1);                            // removes a layer at offset
$layers->has(2);                               // test is a layer is present
```

You can also manipulate them like arrays :

```
<?php
$image = $imagine->open('image.jpg');
$layers = $image->layers();

$layers[0]->save('layer-0.jpg');           // access a layer
$layers[0] = $imagine->open('image2.jpg'); // set layer at offset 0
$layers[]  = $imagine->open('image3.jpg'); // push a new layer in layers
unset($layers[1]);                         // removes a layer at offset
isset($layers[2]);                         // test is a layer is present
```

Note: Layers can be compared as indexed arrays. You should not use string keys.

3.6.3 Generate Animated gif

Imagine provides a simple way to generate animated gif by manipulating layers :

```
<?php

$image = $imagine->open('image.jpg');

$image->layers()
    ->add($imagine->open('image2.jpg'))
    ->add($imagine->open('image3.jpg'))
    ->add($imagine->open('image4.jpg'))
    ->add($imagine->open('image5.jpg'));

$image->save('animated.gif', array(
```

(continues on next page)

(continued from previous page)

```
    'animated' => true,
));
```

When saving an animated gif, you are only required to use the `animated` option.

There are more options that can customize the output, look at the following example :

```
<?php

$image->save('animated.gif', array(
    'animated'       => true,
    'animated.delay' => 500, // delay in ms
    'animated.loops' => 0,   // number of loops, 0 means infinite
));
```

3.6.4 Animated gif frame manipulation

Resizing an animated cats.gif file :

```
<?php
$image = $imagine->open('cats.gif');

$image->layers()->coalesce();
foreach ($image->layers() as $frame) {
    $frame->resize(new Box(100, 100));
}

$image->save('resized-cats.gif', array('animated' => true));
```

The layers (frames) should be coalesced so that they are all in line with each other. Otherwise you may end up with strange artifacts due to how animated GIFs *can* work. Without going into too much detail, think of it as each frame being a patch to the previous one. Also note that not the image, but each frame is resized. This again has to do with how animated GIFs work. Not every frame has to be the full image.

The following example extract all frames of the cats.gif file :

```
<?php

$i = 0;
foreach ($imagine->open('cats.gif')->layers() as $layer) {
    $layer->save("frame-$i.png");
    $i++;
}
```

This one adds some text on frames :

```
<?php

$image = $imagine->open('cats.gif');
$i = 0;
foreach ($image->layers() as $layer) {
    $layer->draw()
          ->text($i, new Font('coolfont.ttf', 12, $image->palette()->color('white')),␣
↪new Point(10, 10));
    $i++;
```

(continues on next page)

(continued from previous page)

```
}

// save modified animation
$image->save('cats-modified.gif', array('flatten' => 'false'));
```

3.7 Image effects

Imagine also provides a fully-featured effects API. To use the api, you need to get an effect instance from you current image instance, using `ImageInterface::effects()` method.

3.7.1 Example

```
<?php

$image = $imagine->open('portrait.jpeg');

$image->effects()
    ->negative()
    ->gamma(1.3);

$image->save('negative-portrait.png');
```

The above example would open a "portrait.jpeg" image, invert the colors, then corrects the gamma with a parameter of 1.3 then saves it to a new file "negative-portrait.png".

Note: As you can notice, all effects are chainable.

3.7.2 Effects API

The current Effects API currently supports these effects :

Negative

The negative effect inverts the color of an image :

```
<?php

$image = $imagine->open('portrait.jpeg');

$image->effects()
    ->negative();

$image->save('negative-portrait.png');
```

Gamma correction

Apply a gamma correction. It takes one float argument, the correction parameter.

```php
<?php

$image = $imagine->open('portrait.jpeg');

$image->effects()
    ->gamma(0.7);

$image->save('negative-portrait.png');
```

Grayscale

Create a grayscale version of the image.

```php
<?php

$image = $imagine->open('portrait.jpeg');

$image->effects()
    ->grayscale();

$image->save('grayscale-portrait.png');
```

Colorize

Colorize the image. It takes one `Imagine\Image\Palette\Color\ColorInterface` argument, which represents the color applied on top of the image.

This feature only works with the Gd and Imagick drivers.

```php
<?php

$image = $imagine->open('portrait.jpeg');

$pink = $image->palette()->color('#FF00D0');

$image->effects()
    ->colorize($pink);

$image->save('pink-portrait.png');
```

Blur

Blur the image. It takes a string argument, which represent the sigma used for Imagick and Gmagick functions (defaults to 1).

```php
<?php

$image = $imagine->open('portrait.jpeg');

$image->effects()
    ->blur(3);

$image->save('blurred-portrait.png');
```

Note: Sigma value has no effect on GD driver. Only GD's IMG_FILTER_GAUSSIAN_BLUR filter is applied instead.

Brightness

Bright or darken the image. It takes an integer argument, which represent the light to add or remove Give a value from -100 to 100, from black to white

```
<?php

$image = $imagine->open('portrait.jpeg');

$image->effects()
    ->brightness(20); //Will bright the image

$image->save('bright-portrait.png');
```

3.8 Image filters and delayed processing

`ImageInterface` in Imagine is very powerful and provides all the basic transformations that you might need, however, sometimes it might be useful to be able to group all of them into a dedicated object, that will know which transformations, in which sequence and with which parameters to invoke on the `ImageInterface` instance. For that, Imagine provides `FilterInterface` and some basic filters that call transformation on the `ImageInterface` directly as an example.

Note: more filters and advanced transformations is planned in the nearest future

3.8.1 Image Transformations, aka Lazy Processing

Sometimes we're not comfortable with opening an image inline, and would like to apply some pre-defined operations in the lazy manner.

For that, Imagine provides so-called image transformations.

Image transformation is implemented via the `Filter\Transformation` class, which mostly conforms to `ImageInterface` and can be used interchangeably with it. The main difference is that transformations may be stacked and performed on a real `ImageInterface` instance later using the `Transformation::apply()` method.

Example of a naive thumbnail implementation:

```
<?php

$transformation = new Imagine\Filter\Transformation();

$transformation->thumbnail(new Imagine\Image\Box(30, 30))
    ->save('/path/to/resized/thumbnail.jpg');

$transformation->apply($imagine->open('/path/to/image.jpg'));
```

The result of `apply()` is the modified image instance itself, so if we wanted to create a mass-processing thumbnail script, we would do something like the following:

```
<?php

$transformation = new Imagine\Filter\Transformation();

$transformation->thumbnail(new Imagine\Image\Box(30, 30));

foreach (glob('/path/to/lots/of/images/*.jpg') as $path) {
    $transformation->apply($imagine->open($path))
        ->save('/path/to/resized/'.md5($path).'.jpg');
}
```

The `Filter\Transformation` class itself is simply a very specific implementation of `FilterInterface`, which is a more generic interface, that lets you pre-define certain operations and variable calculations and apply them to an `ImageInterface` instance later.

3.8.2 Filter Application Order

Normally filters are applied in the order that they are added to the transformation. However, sometimes we want certain filters to always apply first, and others to always apply last, for example always apply a crop before applying a border. You can do this by specifying a priority when passing a filter to the `add()` method:

```
<?php

$transformation = new Imagine\Filter\Transformation();

$transformation->add(new Filter\Basic\Crop($point, $size), -10); //this filter has␣
↪priority -10 and applies early
$transformation->add(new Filter\Advanced\Border($color), 10); //this filter has␣
↪priority 10 and applies late
$transformation->add(new Filter\Basic\Rotate($angle)); //this filter has default␣
↪priority 0 and applies in between

//filters with equal priority will still be applied in the order they were added
```

This is especially useful when you add filters based on user-input.

3.8.3 Filters

As we already know, `Filter\Transformation` is just a very special case of `Filter\FilterInterface`.

Filter is a set of operations, calculations, etc., that can be applied to an `ImageInterface` instance using `Filter\FilterInterface::apply()` method.

Right now only basic filters are available - they simply forward the call to `ImageInterface` implementation itself, more filters coming soon...

3.9 Handling exceptions

Imagine is good with exceptions, in fact, it will throw a lot of them for every possible thing that goes wrong. There are no methods that return `false` on failure, its all exception based.

3.9.1 Exception interface

Every exception class in Imagine implements Exception (`Imagine\Exception\Exception`) interface, making it possible to catch all Imagine exceptions without catching anything not Imagine specific.

```
<?php

try {
    $imagine = new Imagine\Gd\Imagine();

    $imagine->open('/path/to/image.jpg')
        ->thumbnail(new Imagine\Image\Box(50, 50))
        ->save('/path/to/image/thumbnail.png');
} catch (Imagine\Exception\Exception $e) {
    // handle the exception
}
```

This is too generic however and might not work for everyone.

3.9.2 Exception classes

In Imagine, each exception class is extending one of the SPL exception classes, so even if you simply handle SPL exception, Imagine should fit right in. For example `Imagine\Exception\InvalidArgumentException` class extends `InvalidArgumentException`, letting you catch it as an SPL exception or by catching its instance specifically

Note: This technique came from Zend Framework 2

CHAPTER 4

Api docs:

Find them in the API browser

CHAPTER 5

A couple of words in defense

After reading the documentation and working with the library for a little while, you might be wondering "Why didn't he keep width and height as simple integer parameters in every method that needed those?" or "Why is x and y coordinates are an object called Point?". These are valid questions and concerns, so let me try to explain why:

Type-hints and validation - instead of checking for the validity of width and height (e.g. positive integers, greater than zero) or x, y (e.g. non-negative integers), I decided to move that check into constructor of `Box` and `Point` accordingly. That means, that if something passes the type-hint - a valid implementations of `BoxInterface` or `PointInterface`, it is already valid.

Utility methods - a lot of functionality, like "determine if a point is inside a given box" or "can this box fit the one we're trying to paste into it" is also to be shared in many places. The fact that these primitives are objects, lets me extract all of that duplication.

Value objects - as you've noticed neither `BoxInterface` nor `PointInterface` along with their implementations define any setter. That means the state of those objects is immutable, so there aren't side-effects to happen and the fact that they're passed by reference, will not affect their values.

It's OOP man, come on - nothing to add here, really.

CHAPTER 6

Indices and tables

- genindex
- search