Go: Working with Graphics

An image manipulation development guide

Tit Petric

Step by step guide for working with images in Go.

Go With Graphics

Tit Petric

This book is for sale at http://leanpub.com/go-with-graphics

This version was published on 2021-01-02

Also By Tit Petric

Contents

Introduction

About me

I’m passionate about API development, good practices, performance optimizations and educating people to strive for quality. I have about two decades of experience writing and optimizing software, and often solve real problems in various programming languages, Go being my favorite.

You might know some of my work from:

- Author of API Foundations in Go[1],
- Author of 12 Factor Applications with Docker and Go[2],
- Author of Advent of Go Microservices[3],
- Blog author on scene-si.org[4]

Professionally I specialize in writing APIs in the social/media industry and for various content management products. Due to the public exposure of such APIs, it’s performance characteristics are of paramount importance. I’m a solid MySQL DBA with experience on other databases as well.

Who is this book for?

Besides having an use in professional applications, image manipulation, or graphics in general, is an interesting area of work. We will focus on image manipulations in a practical context, implementing an image resizer service, as well as implement serveral filters and other types of color based image manipulations.

If you ever wanted to implement your own image filter or modify images in other fun ways, this book is for you!

[1]https://leanpub.com/api-foundations
[2]https://leanpub.com/12fa-docker-golang
[3]https://leanpub.com/go-microservices
[4]https://scene-si.org

In the book, I will cover these subjects:

1. Standard library image packages
2. Resize and crop
3. Color theory
4. Edge detection
5. Focal point detection
6. Color manipulations (filters)
7. Image overlays, watermarking
8. Image generating
9. Getting creative

How should I study it?

Just go through the book from the start to finish. If possible, try to do the exercises yourself not by copy pasting but by actually writing the code and snippets in the book, tailored to how you would lay out your project.

The work in individual chapters builds on what was done in the previous chapter. The examples are part of a step by step, chapter by chapter process.

Be sure to follow the Requirements section as you're working with the book.

Requirements

This is a book which gives you hands on instructions on image manipulation tasks. We will be using a modern stack of software that is required to complete all the exercises.

Linux and Docker

The examples of the book rely on a recent docker-engine installation on a Linux host. Linux, while a soft requirement, is referenced many times in the book with examples on how to download software, or how we are building our services. You could do this on a Mac, but you might have to adjust some things as you move through the chapters.

- Go 1.14+
- Docker (a recent version),
- Drone CI,
- Various shell utilities and programs (awk, bash, sed, find, ls, make,...)

Please refer to the official docker installation instructions[5] on how to install a recent docker version, or install it from your package manager.

Own hardware

The recommended configuration if you have your own hardware is:

- 2 CPU core, x86/amd64, 64bit,
- 2GB ram,
- 128GB disk (SSD)

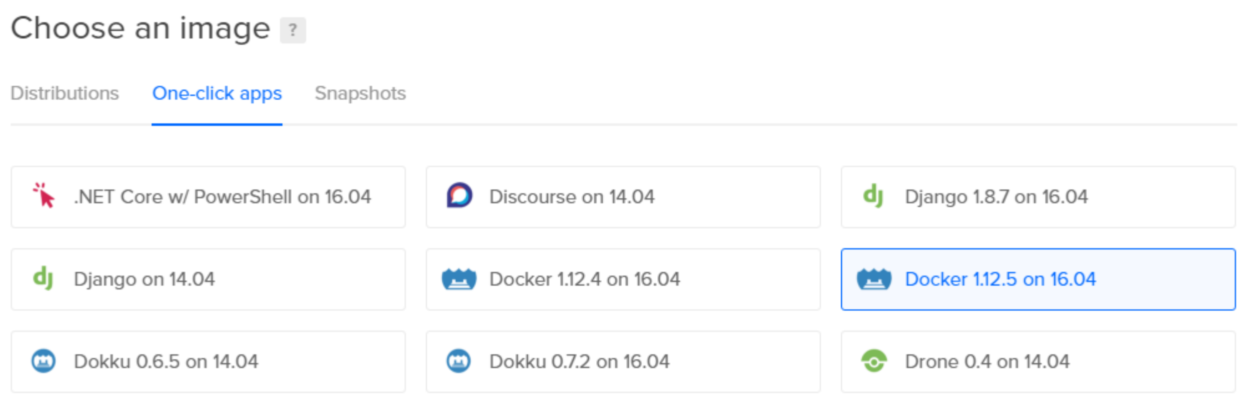
The minimal configuration known to mostly work is about half that, but you might find yourself in a tight place as soon as your usage goes up. If you're just tying out docker, a simple virtual machine might be good enough for you, if you're not running Linux on your laptop already.

[5]https://docs.docker.com/engine/installation/linux/

Cloud quick-start

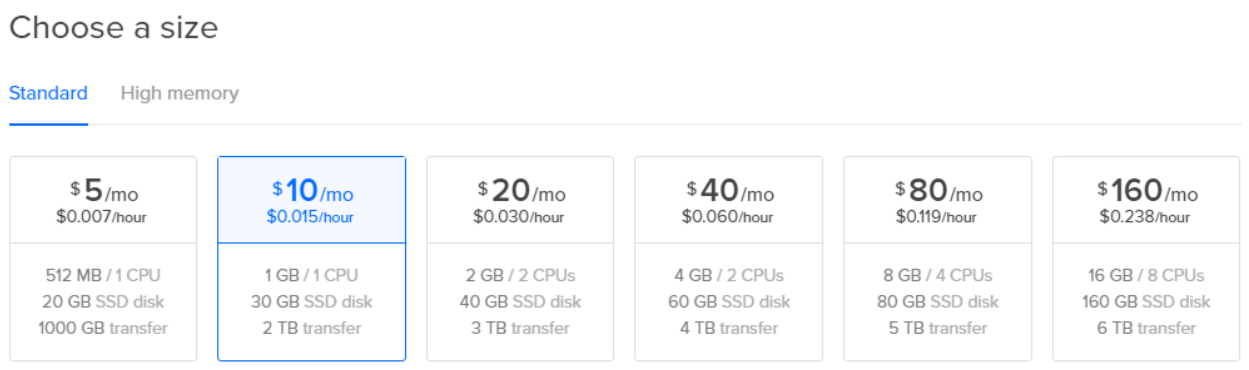
If having your own hardware is a bit of a buzzkill, welcome to the world of the cloud. You can literally set up your own virtual server on Digital Ocean within minutes. You can use this DigitalOcean referral link[6] to get a $10 credit, while also helping me take some zeros of my hosting bills.

After signing up, creating a Linux instance with a running Docker engine is simple, and only takes a few clicks. There's this nice green button on the top header of the page, where it says "Create Droplet". Click it, and on the page it opens, navigate to "One-click apps" where you choose a "Docker" from the list.



Choose Docker from "One-click apps"

Running docker can be disk-usage intensive. Some docker images may "weigh" up to or more than 1 GB. I would definitely advise choosing an instance with *at least* 30GB of disk space, which is a bargain for $10 a month, but you will have to keep an eye out for disk usage. It's been known to fill up.


Choose a size

Standard High memory

$5/mo	$10/mo	$20/mo	$40/mo	$80/mo	$160/mo
$0.007/hour	$0.015/hour	$0.030/hour	$0.060/hour	$0.119/hour	$0.238/hour
512 MB / 1 CPU	1 GB / 1 CPU	2 GB / 2 CPUs	4 GB / 2 CPUs	8 GB / 4 CPUs	16 GB / 8 CPUs
20 GB SSD disk	30 GB SSD disk	40 GB SSD disk	60 GB SSD disk	80 GB SSD disk	160 GB SSD disk
1000 GB transfer	2 TB transfer	3 TB transfer	4 TB transfer	5 TB transfer	6 TB transfer

Choose a reasonable disk size

Aside for some additional options on the page, like chosing a region where your droplet will be running in, there's only a big green "Create" button on the bottom of the page, which will set up everything you need.

[6] https://m.do.co/c/021b61109d56

Standard library image packages

The first step in working with images in Go is using the standard library package image[7]. To use the package, you have to register an image decoder. The standard library provides decoders for only the most standard image types - `jpg`, `png`, and `gif`.

The standard library also provides subpackages `image/color` for handling colors and `image/draw` to layer images one over the other.

Loading images

Let's start by writing a CLI for `imageinfo`. We want to take an input file, decode it (load the image), and then print image dimensions. Create the `cmd/imageinfo` path and create the following files in that location.

We will start with this simple entrypoint (`main.go`):

```go
package main

import (
  _ "image/gif"
  _ "image/jpeg"
  _ "image/png"
  "log"
  "os"
)

func main() {
  if len(os.Args) < 2 {
    log.Fatalln("Usage: imageinfo [image.jpg/png/gif]")
  }

  if err := printInfo(os.Args[1]); err != nil {
    log.Fatalln(err)
  }
}
```

[7]https://pkg.go.dev/image?tab=doc

The entrypoint takes care of loading the image drivers which we will support. By prefixing the import path with _, only the image drivers get registered, the rest of the packages are unused in this file.

The main() functions only takes care of ensuring that a single parameter was passed to our program. In case the parameter isn't present, a descriptive error will be printed. We invoke `printInfo` with the filename to read.

Next, we need to create a helper function to load images. Let's create `load.go` with the following contents:

```go
package main

import (
  "image"
  "os"
)

func load(filename string) (image.Image, error) {
  f, err := os.Open(filename)
  if err != nil {
    return nil, err
  }
  defer f.Close()

  m, _, err := image.Decode(f)
  return m, err
}
```

The load() function only takes care of loading the image from a file. It passes the file reader to image.Decode, and returns any error that might have occured.

Now, for each image, we would like to print two parts of information. We would like to know the image dimensions, and we would like to know what the average color of the image is.

Create a `info.go` to implement our `imageInfo` function:

```go
package main

func printInfo(filename string) error {
  m, err := load(filename)
  if err != nil {
    return err
  }

  printDimensions(m)
  printAverageColor(m)
  return nil
}
```

Getting the dimensions of the image is done by retrieving the image bounds, and getting the information from the returned image.Rectangle[8]. Go ahead and create a `dimensions.go` file:

```go
package main

import (
  "fmt"
  "image"
)

func printDimensions(m image.Image) {
  bounds := m.Bounds()
  fmt.Printf("Width:  %d\n", bounds.Dx())
  fmt.Printf("Height: %d\n", bounds.Dy())
}
```

We used `bounds.Dx()` and `bounds.Dy()` as the shorthand calculations for width and height of the image. If we need image dimensions, we can use these built-in functions on `image.Rectangle`.

Reading image colors

For our next task, we want to calculate the average color of the image. This means reading all the image pixels, adding all the colors together, and finally dividing by the number of pixels in the image.

We will use another Image interface function for this:

[8]https://golang.org/pkg/image/#Rectangle

```go
func At(x, y int) color.Color
```

The `Color` interface provides a `RGBA() (r,g,b,a uint32)`, with 16-bit accuracy, meaning each value ranges between [0, 0xFFFF] (16 bits). The `uint32` type has been chosen to avoid an overflow when a color is multiplied by the blend factor up to 0xFFFF (16 bits). The full color is encoded in a 64 bit value, with 16 bits per component.

Let's create a `printAverageColor(m image.Image)` function to print the average RGBA for the whole image. We need to read each pixel color value, add them together, and then divide by pixel count.

While the bounds have helpers for getting the width and height of the image, the actual image boundaries may start at values different from `X=0,Y=0`. This means we need to iterate the pixels by starting with the bounds Min.X and Min.Y values.

```go
package main

import (
  "fmt"
  "image"
)

func printAverageColor(m image.Image) {
  var r, g, b, a uint64

  bounds := m.Bounds()
  for y := bounds.Min.Y; y < bounds.Max.Y; y++ {
    for x := bounds.Min.X; x < bounds.Max.X; x++ {
      cr, cg, cb, ca := m.At(x, y).RGBA()
      r += uint64(cr)
      g += uint64(cg)
      b += uint64(cb)
      a += uint64(ca)
    }
  }

  // multiply with 256 to get 8 bits per component after division
  count := uint64(bounds.Dx() * bounds.Dy() * 256)

  fmt.Printf("R: %d\nG: %d\nB: %d\nA: %d\n", r/count, g/count, b/count, a/count)
}
```

We can invoke this function from `printImageInfo`, and look at the output:

```
Width:  640
Height: 427
R: 80
G: 75
B: 68
A: 255
```

As I'm loading a `jpg` file for testing, there is no alpha channel on it, meaning every color is opaque. If this was a `png` with a defined alpha channel, the story would be a little bit different:

A white pixel with 50% opacity results in the following:

```
Width:  1
Height: 1
R: 127
G: 127
B: 127
A: 127
```

Here you can note that the color returned is already multiplied by the alpha channel. The color package documents this:

> RGBA returns the alpha-premultiplied red, green, blue and alpha values for the color. [...] An alpha-premultiplied color component c has been scaled by alpha (a), so has valid values 0 <= c <= a.

This is dependant on what kind of image we have loaded.

Image color space

Depending on the image type, we may have loaded a paletted image (gif), or an 8-bit greyscale image. Each of these image examples has a particular color model, which we can inspect with the third and final function implemented by the Image interface.

```go
func ColorModel() color.Model
```

In order to ensure that we're dealing with a particular color model, we must create a new image of a particular type (NRGBA) and then draw the source image onto that. Think of it like copy-pasting a grey-scale image into a true-color empty image, and then saving it as true-color.

This brings us to the `image/draw` package, and it's `Draw()` function.

Let's implement converters to the `NRGBA` and `NRGBA64` image types. By using the concrete image type, we can get other functions that also allow us to set/write pixels to the image, something that the Image interface doesn't provide us. Create `convert.go`:

```go
package main

import (
  "image"
  "image/draw"
)

func toNRGBA(in image.Image) *image.NRGBA {
  bounds := in.Bounds()
  out := image.NewNRGBA(image.Rect(0, 0, bounds.Dx(), bounds.Dy()))
  draw.Draw(out, out.Bounds(), in, bounds.Min, draw.Src)
  return out
}

func toNRGBA64(in image.Image) *image.NRGBA64 {
  bounds := in.Bounds()
  out := image.NewNRGBA64(image.Rect(0, 0, bounds.Dx(), bounds.Dy()))
  draw.Draw(out, out.Bounds(), in, bounds.Min, draw.Src)
  return out
}
```

Now, using the NRGBA64 image won't give us NRGBA64 colors when calling the image At() function. The particular function will *always* return a premultiplied color. The NRGBA64 struct provides another function, `NRGBA64At`, which returns the color without premultiplication.

We need to read individual fields from the NRGBA64 color struct. If we would invoke RGBA(), it will return the multiplied color components. As we don't need 16 bit precision, I'll be using the `NRGBA` type here.

Let's rewrite printAverageColor a bit, to split it into two functions. The updated function body to get the average color looks like this:

```go
func getAverageColor(in image.Image) *color.NRGBA {
  var r, g, b, a uint64

  m := toNRGBA(in)

  bounds := m.Bounds()
  for y := bounds.Min.Y; y < bounds.Max.Y; y++ {
    for x := bounds.Min.X; x < bounds.Max.X; x++ {
      color := m.NRGBAAt(x, y)
      r += uint64(color.R)
      g += uint64(color.G)
      b += uint64(color.B)
      a += uint64(color.A)
    }
  }

  count := uint64(bounds.Dx() * bounds.Dy())
  return &color.NRGBA{
    R: uint8(r / count),
    G: uint8(g / count),
    B: uint8(b / count),
    A: uint8(a / count),
  }
}

func printAverageColor(in image.Image) {
  color := getAverageColor(in)
  fmt.Printf("R: %d\nG: %d\nB: %d\nA: %d\n", color.R, color.G, color.B, color.A)
}
```

We can now re-use `getAverageColor` when we want to get the actual color. Re-running with the 50% white png image gives us the expected output now:

```
Width:  1
Height: 1
R: 255
G: 255
B: 255
A: 127
```

We can now read true colors, which can be used for HTML composed color entities. As RGB(A) colors in CSS aren't pre-multiplied, relying on NRGBA image formats when doing color processing is a must. Whatever image processing APIs you will write, should use a concrete image type like NGRBA64 as soon as possible, do you don't do unnecessary conversions.

I have chosen NRGBA as 8 bits per color component is the most standard image format used in JPEG/PNG files and HTML color codes as well.

Saving images

Saving images is done with their respective image drivers. The interface for saving the images isn't generic, as each driver implements their own options. While JPEG images have a `Quality` parameter (percentage as int), PNG images only provide a `CompressionRatio` option.

Create a `save.go` with our needed imports:

```go
package main

import (
  "errors"
  "image"
  "image/jpeg"
  "image/png"
  "io"
  "os"
  "path"
)
```

We can abstract our image writers behind an interface:

```go
type imageWriter func(io.Writer, image.Image) error

func writeJpeg(w io.Writer, img image.Image) error {
  opt := jpeg.Options{
    Quality: 90,
  }
  return jpeg.Encode(w, img, &opt)
}

func writePng(w io.Writer, img image.Image) error {
  encoder := &png.Encoder{
    CompressionLevel: png.BestSpeed,
  }
  return encoder.Encode(w, img)
}
```

Based on the image filename we're trying to save, we can decide which writer we need to use to save the image to disk:

```go
func saveImage(filename string, img image.Image, writer imageWriter) error {
  f, err := os.Create(filename)
  if err != nil {
    return err
  }
  defer f.Close()

  return writer(f, img)
}
```

```go

func save(filename string, img image.Image) error {
  ext := path.Ext(filename)
  if ext == ".jpg" {
    return saveImage(filename, img, writeJpeg)
  }
  if ext == ".png" {
    return saveImage(filename, img, writePng)
  }
  return errors.New("save: unsupported extension " + ext)
}
```

And that's all there is to it. Implementing a `gif` writer, or any other image format, is left as an exercise to the reader.

Resize and crop

Camera phones (or actual digital cameras) usually produce a variety of resolutions that may be too big to use directly in web or mobile applications. They need to be resized to a smaller resolution, and usually cropped to a variety of formats of different ratios.

With video applications, the desired ratios are usually 16:9 to satisfy modern TVs, while for apps like instagram, a ratio of 1:1 is preffered, so there is part of the screen reserved for image metadata, like it's author, image comment and other widgets.

- A 3840 x 2160 pixel resolution is used for 4K displays,
- A 1920 x 1080 pixel resolution is used for HD ready (1/4th of 4K display),
- A 1080 x 1080 pixel resolution is the default for Instagram posts.

Instagram image width is tailored towards iphone 6-8 screen sizes, which have a width of 1080 pixels. They also have a height of 1920 pixels, which, when in portrait mode, gives a person a HD ready display size.

Interpolation functions for resizing

For an image of an arbitrary width/height, resizing down to a 100x100 image would be done similarly to this *pseudo code* (assume that division returns a floating point):

```
w, h := 1920, 1080
dw, dh := 100, 100
for y := 0; y < dh; y++ {
  for x := 0; x < dw; x++ {
    // calculate source pixels position
    //
    // x / dw = width in % of the target pixel,
    // multiplied by w = source pixel x position

    sx = round((x / dw) * w)
    sy = round((y / dh) * h)

    dest.Set(x, y, source.NRGBAAt(sx, sy))
  }
}
```

How the color is calculated is called an interpolation function. Interpolation is a process where an algorythm calculates the color in some particular way, based on a the source pixel position.

For our particular code sample, the algorythm is called the nearest neighbour. It's called that because all the colors chosen are found in the source image, by calculating the most appropriate x/y values.

The pitfall of this approach is that resizing images will produce jagged destination images, where a significant part of image information is lost. To demonstrate, let's see how a 5x1 pixels image would be resized:

```
source image: [255, 0, 255, 0, 255]
```

Resizing the image from 5 to 4 pixels wide:

```
dest[0] = src[round((0 / dw) * w)] /* index = round(0    * 5 = 0)    = 0, color = 25\
5 */
dest[1] = src[round((1 / dw) * w)] /* index = round(0.25 * 5 = 1.25) = 1, color = 0 \
*/
dest[2] = src[round((2 / dw) * w)] /* index = round(0.5  * 5 = 2.5) = 3, color = 0 */
dest[3] = src[round((3 / dw) * w)] /* index = round(0.75 * 5 = 3.75) = 4, color = 25\
5 */
```

So, the destination image would be `[255, 0, 0, 255]`. When upscaling the image to a larger resolution, we would get what's generally referred to as a "pixelated" image. In popular media, Minecraft is the most common example where pixelated images are used extensively.

When resizing images to a smaller size using the nearest neighbor method, there is obvious data loss compared to the source image. In terms of "optical" difference, interchanged white/black values would seem as grey at a distance. A better interpolation function is able to calculate a better approximation of the resulting colors, producing a more optically-pleasing image.

A trivial method to implement is called bilinear interpolation. It's called bilinear because we need to interpolate two values, the color along the X axis, and the color along the Y axis. For any given (X, Y) input, the fractions (subpixels) are used to calculate the final color.

```
// non-rounded sx/sy
sx = (x / dw) * w
sy = (y / dh) * h

// subpixel positions:
subx = sx % 1.0
suby = sy % 1.0

// source pixel positions:
```

```
top = floor(sy)
left = floor(sx)
right = min(left+1, w-1) // clamp to image size
bottom = min(top+1, h-1) // clamp to image size

// 4 colors for each hard x/y pair
colors := [4]*color.NRGBA{
  source.NRGBAAt(left, top),
  source.NRGBAAt(right, top),
  source.NRGBAAt(left, bottom),
  source.NRGBAAt(right, bottom),
}
```

We have calculated two subpixel fractions, `subx` and `suby`. A `subx` fraction of `0.5` would mean that the source pixel color should be calculated as the average between left/right pixel colors.

Now, the most simple way to produce the final color, is to create a 2D multiplication matrix:

```
// a 0.25 ratio suby is weighted to the top pixel
matrix_top = 1.0 - suby

// a 0.25 ratio subx is weighted to left pixel
matrix_left = 1.0 - subx

// the rest
matrix_right, matrix_left = subx, suby

matrix := [4]float{
  matrix_top + matrix_left,
  matrix_top + matrix_right,
  matrix_bottom + matrix_left,
  matrix_bottom + matrix_right,
}
```

For bilinear interpolation we need to multiply each value in `colors` with corresponding indices in `matrix`. Adding all the values together produces the final interpolated color. Since we are interpolating two sets of pixels, we need a final division by two, to calculate the accurate color value.

With the nearest neighbour method in the previous example, matrix is generally assumed to be `[1, 0, 0, 0]` along each axis, picking only one literal color (top left).

Given a pixel in position `1500.5` and `950.5`, the matrix would be `[0.5, 0.5, 0.5, 0.5]` and the produced color would be an average of all four pixels in the colors[] slice.

```
// pseudo code, but imagine colors[x] to be uint8
color := ((colors[0] * matrix[0]) +
          (colors[1] * matrix[1]) +
          (colors[2] * matrix[2]) +
          (colors[3] * matrix[3])) / 2.0
```

The destination image now becomes `[255, 64, 127, 191]`. Overall, the image is more realistically resized. When using bilinear interpolation for image upscaling, the produced image will seem blurred but will still retain hard pixel edges.

Additional notable interpolation methods include:

1. Bicubic interpolation,
2. Mitchell-Netravali interpolation,
3. Lanczos interpolation,

In terms of upscaling images, the Lanczos interpolation produces the most naturally correct image, as if a larger image was extensively blurred, but kept highlights more color-accurate. It works well for resizing the images down to smaller sizes too, at the cost of being more CPU expensive.

In comparison, when using bicubic interpolation, the highlights are lost, and often, even in comparison with bilinear interpolation, it seems like the upscaled image is created from a pixelated source which was extensively blurred.

While bilinear interpolation produces a color based only on 2 pixels along each axis, other interpolation algorythms above generally use 4 pixels or more (Lanczos can use 6 pixels or even more) to produce a more accurate color. Generally, the more pixels used, the better looking the image is when it's being upscaled.

Resizing images

We will resort to using an existing package to provide our resizing functionality. The package nfnt/resize[9] implements the methods described above, and we will default to use the Lanczos3 interpolation method when we'll resize our images.

When we are dealing with resizing images, we usually resort to one of the following modes when resizing:

1. Letterboxing the destination image,
2. A covering image

The aspect ratio is the ratio between the width and the height of the image. So, a 1920x1080 image has an aspect ratio of 16:9, while a 1000x1000 image has a ratio of 1:1. These are common ratios:

[9] https://github.com/nfnt/resize

- 21:9 - Ultra wide content (used to save vertical space),
- 16:9 - HD video, widescreen monitors/TVs,
- 4:3 - Cameras, classic monitors - landscape,
- 3:4 - Same but in portrait orientation,
- 1:1 - Instagram

While these ratios are pretty standard, the original image sources usually come in 16:9 or 4:3/3:4, depending if the source was either video or a camera, or hopefully, something close to those. When producing website content, often content in various ratios is needed in different parts of the website.

When resizing images the ratio of the source material must to be preserved, so the image doesn't seem like it's stretched. To achieve that, the following two strategies are usually used:

Letterboxing the image

The term "letterbox" comes from the television. When people started shooting film in widescreen formats, they added black bars above and below the video for playback on 4:3 TVs. This was done to keep the original aspect ratio.

In our case, if we want to produce a 640 x 480 pixels image from a 1920 x 1080 pixels image, we need to calculate the resized dimensions of the original image, that will fit into the smaller image.

```
srcWidth, srcHeight := 1920, 1080
dstWidth, dstHeight := 640, 480

// ratioWidth := 0.33333...
ratioWidth := dstWidth / srcWidth

// ratioHeight := 0.44444...
ratioHeight := dstHeight / srcHeight
```

By calculating the ratios for both width and height, we actually calculated two sizing factors. By using the ratioWidth sizing factor, the image would be resized according to fit the destination width:

```
// newWidth := 1920 * 0.33333 = 640
newWidth := srcWidth * ratioWidth
// newHeight := 1080 * 0.33333 = 360
newHeight := srcHeight * ratioWidth
```

By using the ratioHeight sizing factor, the image would be resized to fit the destination height.

```go
// newWidth := 1920 * 0.44444 = 853
newWidth := srcWidth * ratioWidth
// newHeight := 1080 * 0.44444 = 480
newHeight := srcHeight * ratioWidth
```

It's now time to move beyond pseudo code, and implement our resizer. Start by creating a `cmd/resize` folder, and a `main.go` file:

```go
package main

import (
  "flag"
  "log"

  _ "image/gif"
  _ "image/jpeg"
  _ "image/png"
)
```

We need to create our `main()` function, this time by using standard library configuration flag[10] package. We will take four possible flags, the input and output filenames (`-in` and `-out` respectively), and the width/height of the destination image.

```go
func main() {
  var (
    input  string
    output string
    width  uint
    height uint
  )
  flag.StringVar(&input, "in", "", "Input filename")
  flag.StringVar(&output, "out", "", "Output filename")
  flag.UintVar(&width, "width", 640, "Output width")
  flag.UintVar(&height, "height", 480, "Output height")
  flag.Parse()

  if err := loadAndResize(output, input, width, height); err != nil {
    log.Fatalln(err)
  }
}
```

Let's now create the `loadAndResize` function in `resize.go`:

[10]https://golang.org/pkg/flag/

```go
package main

import (
  "errors"
)

func loadAndResize(output string, input string, width uint, height uint) error {
  if input == "" {
    return errors.New("missing argument: input filename")
  }
  if output == "" {
    return errors.New("missing argument: output filename")
  }
  if width <= 0 || height <= 0 {
    return errors.New("invalid argument: width/height")
  }

  img, err := load(input)
  if err != nil {
    return err
  }

  out := resizer(img, width, height)

  return save(output, out)
}
```

The function validates our inputs, loads the image, and saves the image. And what remains is implementing our `resizer()` function (`resizer.go`):

```go
package main

import (
  "image"
  "math"

  "github.com/nfnt/resize"
)

func resizer(img image.Image, width uint, height uint) image.Image {
  b := img.Bounds()
  srcWidth, srcHeight := float64(b.Dx()), float64(b.Dy())

  ratioWidth := float64(width) / srcWidth
  ratioHeight := float64(height) / srcHeight

  dim := func(width, height, ratio float64) (uint, uint) {
    var (
      newWidth  = uint(math.Round(width * ratio))
      newHeight = uint(math.Round(height * ratio))
    )
    return newWidth, newHeight
  }

  var resized image.Image
  w, h := dim(srcWidth, srcHeight, ratioWidth)
  if h > height {
    w, h = dim(srcWidth, srcHeight, ratioHeight)
  }
  resized = resize.Resize(w, h, img, resize.Lanczos3)

  return resized
}
```

Particularly, the resizer function only handles resizing the image to keep aspect ratio and fit inside the target dimensions. We need to add some handling code that will add the letterbox around the image to fill out the requested size.

Let's create a `letterbox.go` file:

```go
package main

import (
  "image"
  "image/color"
  "image/draw"
)

func letterbox(img image.Image, uwidth uint, uheight uint) image.Image {
  resized := resizer(img, uwidth, uheight)
  b := resized.Bounds()
  width, height := int(uwidth), int(uheight)
  w, h := b.Dx(), b.Dy()

  if w < width || h < height {
    offsetX, offsetY := (width-w)/2, (height-h)/2
    dest := image.NewRGBA(image.Rect(0, 0, width, height))

    // fill image with a solid color (blue)
    fillColor := color.NRGBA{0, 0, 255, 255}
    draw.Draw(dest, dest.Bounds(), &image.Uniform{fillColor}, image.Point{}, draw.Sr\
c)

    // draw resized image with offset
    destStart := image.Pt(offsetX, offsetY)
    destRect := image.Rectangle{destStart, destStart.Add(resized.Bounds().Size())}
    draw.Draw(dest, destRect, resized, image.Point{}, draw.Src)

    return dest
  }

  return resized
}
```

There are two things to note here:

We are filling the initial image to be letterboxed by using image.Uniform{}[11]. The struct represents an infinite-sized image of an uniform color.

We are drawing the resized image with a `offsetX` or `offsetY` offset, producing either a vertical or horizontal letterbox, depending on the source image ratio.

Since we already implemented getting the average color in the previous chapter, let's use the images average color to fill out the letterbox, instead of an aggresive blue color. Change the fillColor to take the average color value from the image:

```
fillColor := getAverageColor(img)
```

In case of nearly solid color images, the output is indistinguishable from having a larger image source. In other images, it's reasonable to asume that the average color will mimic the general tone of the image, so pictures of the sky will end up with an off-white blue tone, while pictures from nature might end up with pastel/earth tones. Depending on the source image color vibrancy, better strategies can be used.

[11]https://golang.org/pkg/image/#Uniform