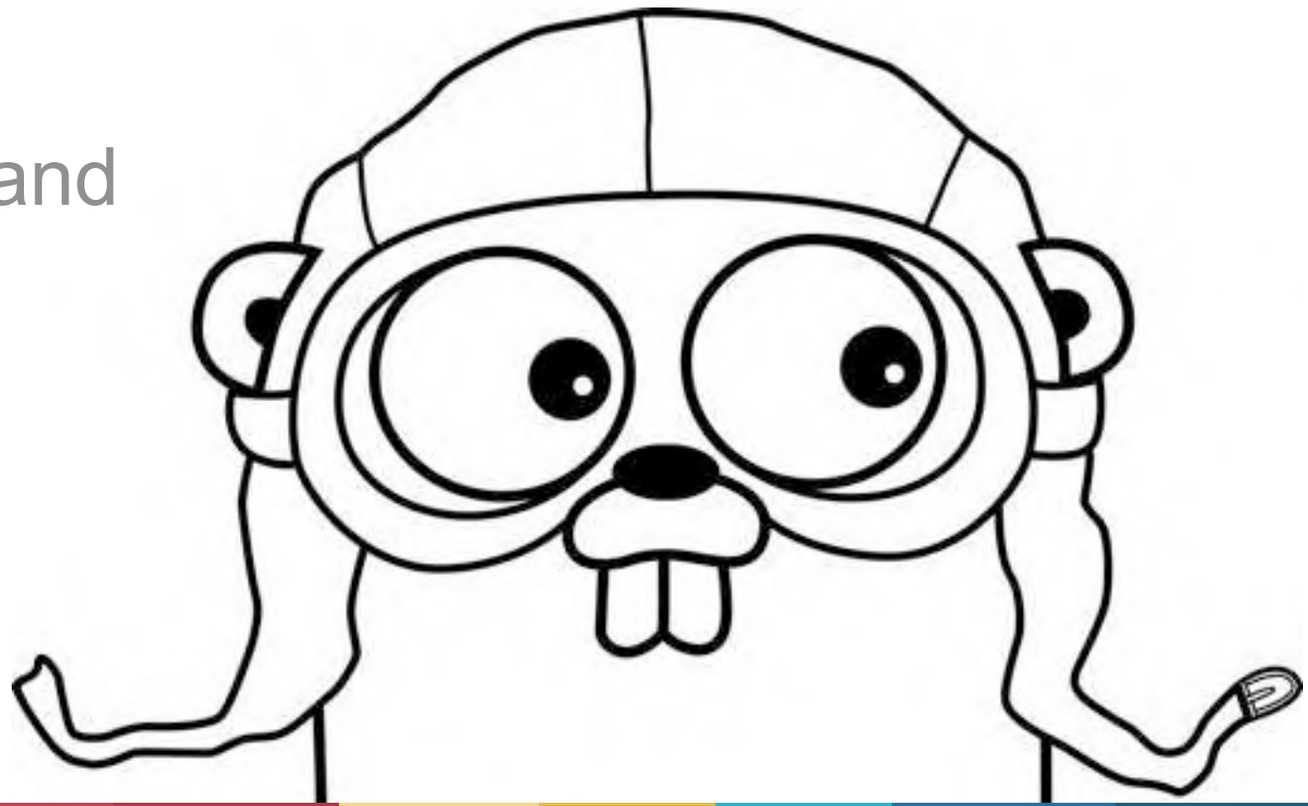

Google™
11
I/O

Writing Web Apps in Go

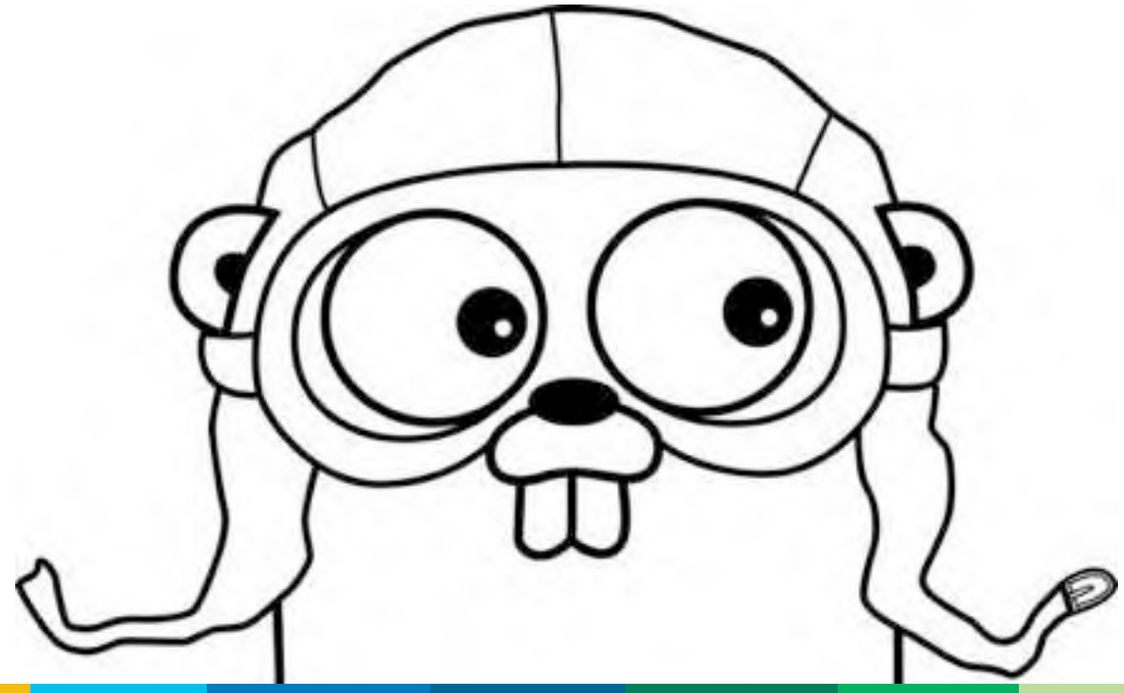
Andrew Gerrand
Rob Pike
May 10, 2011



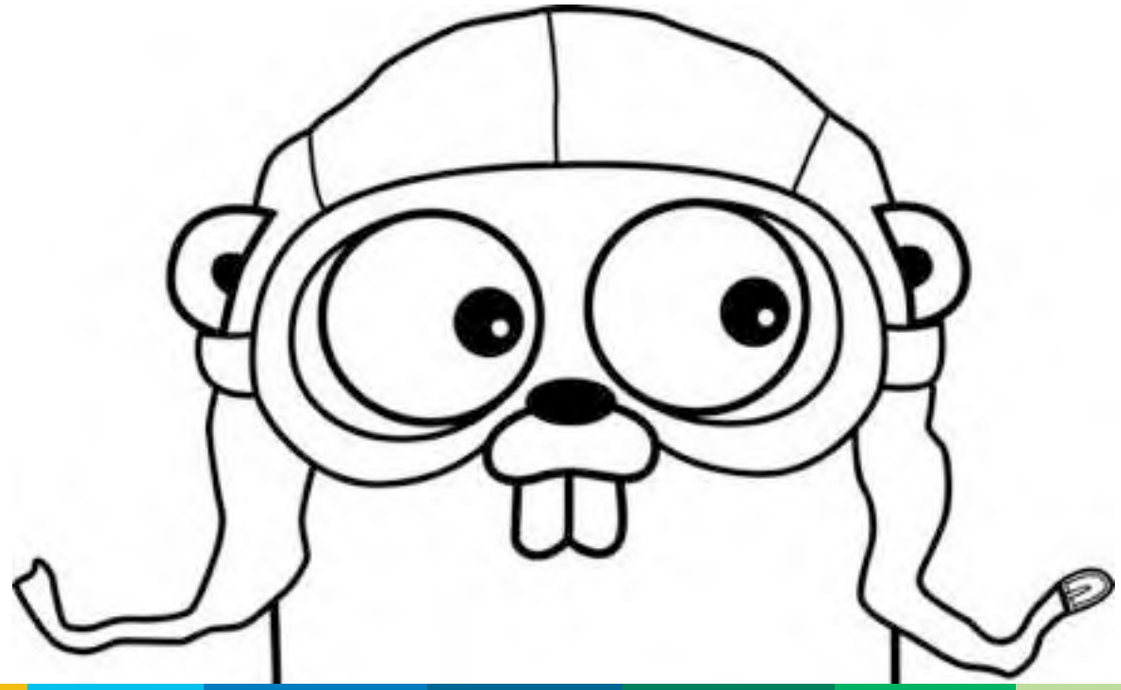
Google I/O 11

Feedback

http://goo.gl/U5rYj
#io2011 #DevTools



What is Go?



A programming language for today's world

- Go is fun, efficient, and open source.
- Launched in November, 2009.
- Goals:
  - fun to work in.
  - efficient - performance approaching C for CPU-intensive tasks,
  - designed for modern hardware (multicore, networks),
  - aimed at software such as web servers,
  - ... but turned out to be a great general-purpose language.

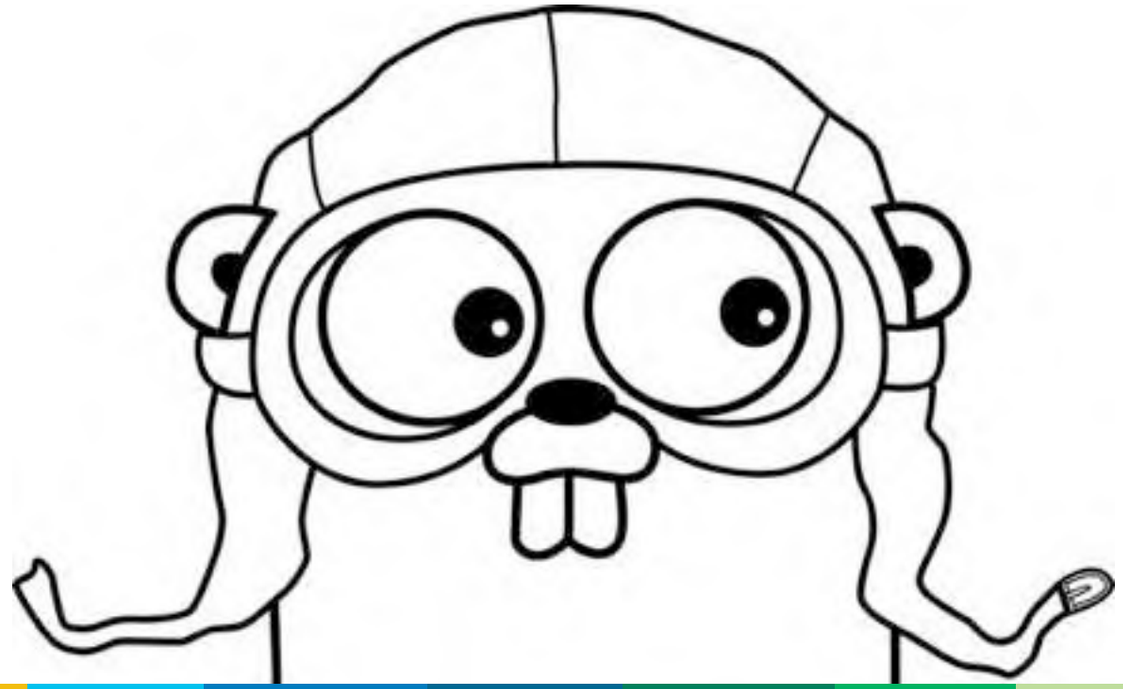
- True open source
  - all development is in the open,
  - many contributors from around the world.

Overview

- News from a year of development.
- A web server written in Go.
- Deploying Go in production.
- Prerequisite: some knowledge of web servers, not much Go.

- Some big announcements and fun giveaways.
  - Be sure to stay until the end.

A year's progress



Ports

- Windows supported!
  - Our compiler ported, thanks to our open source community, especially:
    - Alex Brainman, Hector Chu, Joe Poirier, 韦光京 (Wei Guangjing).
  - Express Go (an interpreter with JIT compilation).
  - Ergo (a new compiler for Windows; under development).
- ARM port (5g) is solid,
- GCC4.6 release includes gccgo,
- SWIG,
- and more....

GDB has Go support

```
(gdb) run
Starting program: /home/r/6.out
Breakpoint 1, runtime.throw (s=void) at runtime.c:97
97 runtime·throw(int8 *s)
(gdb) info goroutines
  2  waiting runtime.gosched
  1  waiting runtime.gosched
(gdb) goroutine 1 where
...
#3  0x00400c2d in main.f (myChan=0x0cba0) at bug.go:4
#4  0x0400c9a in main.main () at bug.go:9
...
(gdb) goroutine 2 where
...
#3  0x0400c2d in main.f (myChan=0x0c000) at bug.go:4
... in ?? ()
(gdb)
```

Language simplified

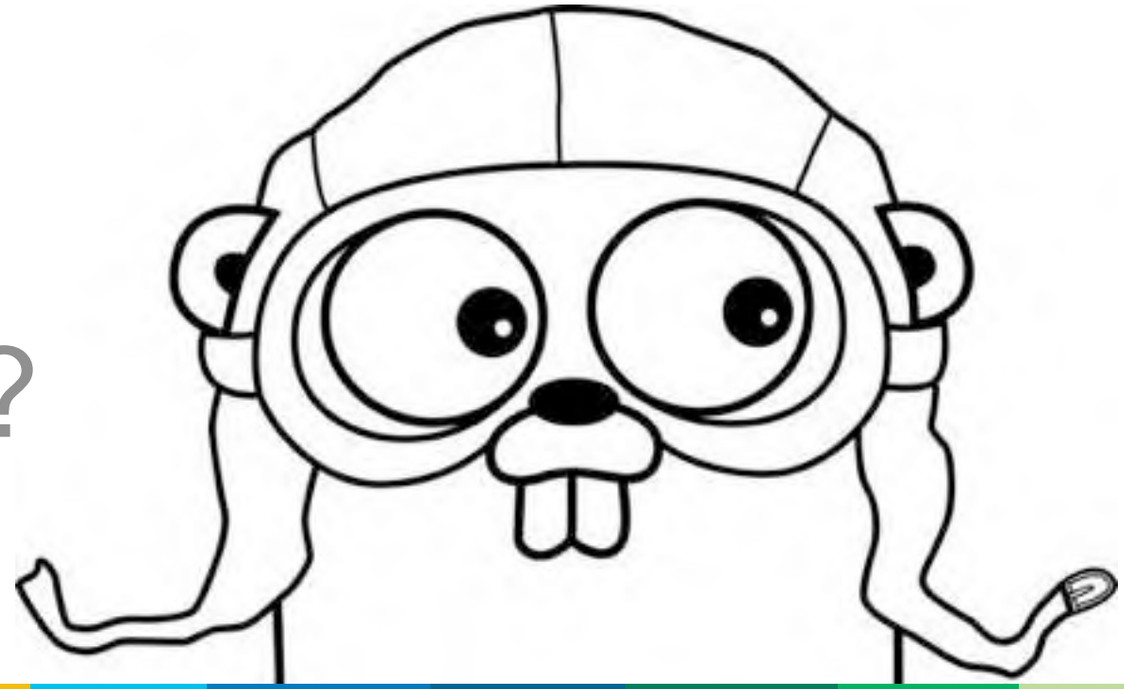
- Non-blocking communication operations gone.
  - `select` statement can do them, more cleanly.
- `closed` built-in gone.
  - Too subtle and race-prone.
- Unsized `float`, `complex` gone.
  - use `float32`, `complex64`, etc.
- Slices simplified.
- Variadic functions easier: `...Type`
  - Also `slice...` at call site.
- `append` built-in added.
- Nicer composite literals.

Open source

- Already mentioned Windows port.
- More than a thousand change sets from over 130 (mostly external) contributors.
- InfoWorld Bossie award:

Google's Go language tries to restore simplicity to programming. It does away with numerous constructs that have crept into all OO languages by re-imagining how to simplify and improve the conversation between a developer and the code. Go provides garbage collection, type safety, memory safety, and built-in support for concurrency and for Unicode characters. In addition, it compiles (fast!) to binaries for multiple platforms. Go ... shows a new, exciting direction in programming languages.

A systems language?



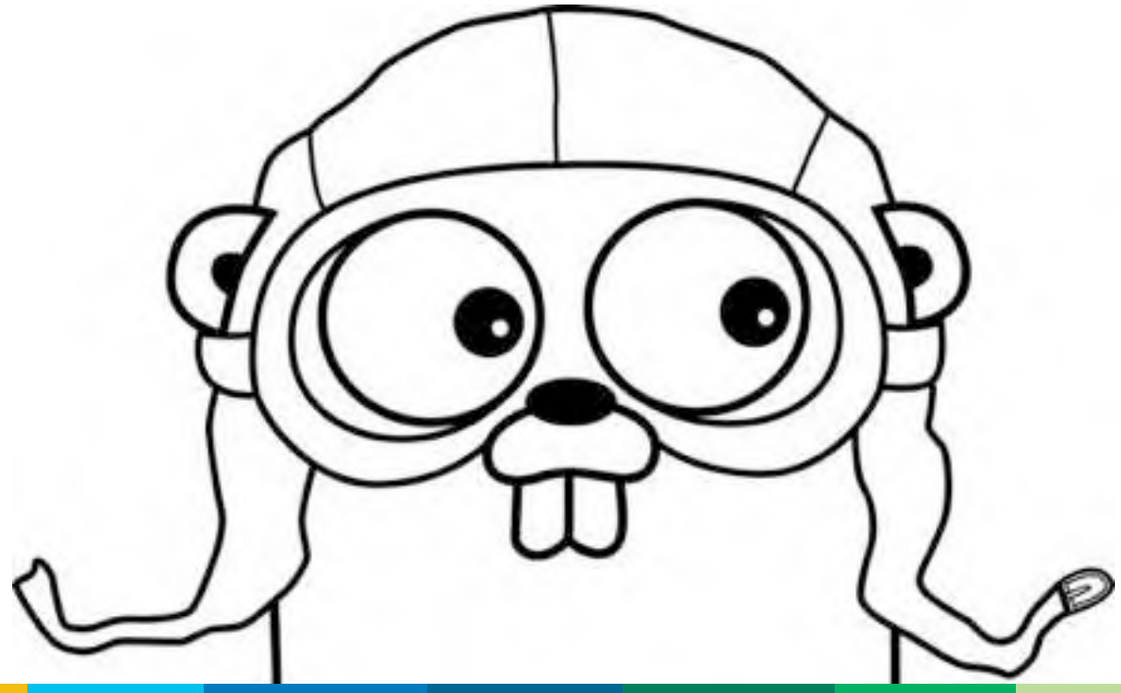
What sort of language is Go?

- Earliest announcements: "A systems language."
- This confused people.

- Doesn't matter! Go is a good general-purpose language that happens to be fun and fast, scalable and efficient.

- A great combination for writing lots of things, including
  - systems like web servers...
    - such as this one...

A web server



“Moustachio”

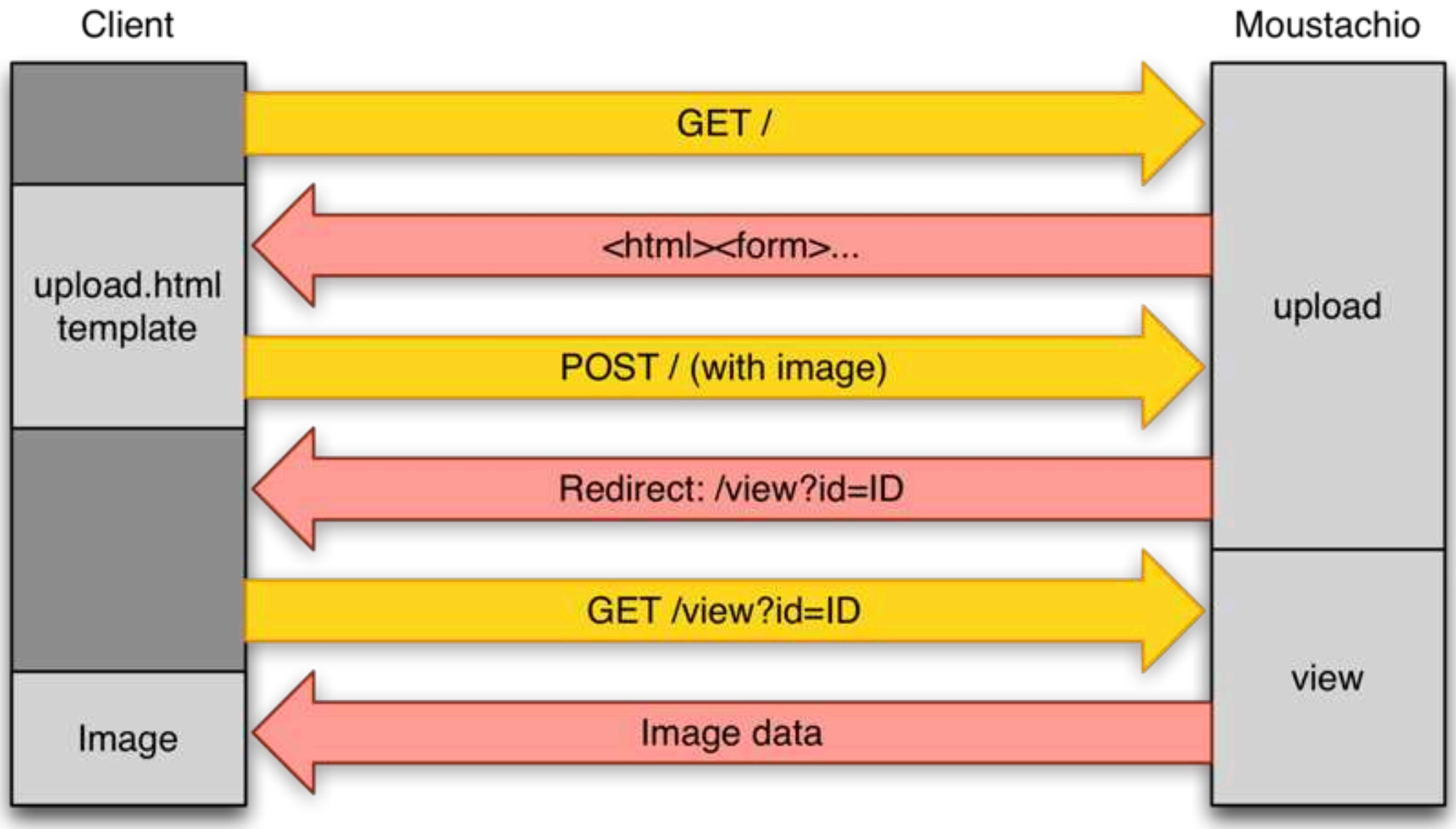
- Our bid to enter the burgeoning photo sharing market.
- Core functionality:
  - upload a photo of something,
  - draw a moustache on it,
  - share it with friends.
- This will demonstrate Go’s abilities to:
  - serve web pages (using HTML templates),
  - receive uploaded files via HTTP,
  - work with the file system,
  - decode, manipulate, and encode images, and
  - interact with OAuth-authenticated RESTful APIs.

Hello, web

The http package provides the basics for serving web pages.

```
package main

import "fmt"
import "http"

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, world")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Data flow: uploading and viewing images



File Uploads (I): define an HTML form

To receive a file from a client, serve an HTML form with an `input` element of type `file`, which we call `upload.html`:

```
<html>
<head>
    <title>Moustachio</title>
</head>
<body>
    <p>Upload an image to moustachify:</p>
    <form action="/" method="POST"
          enctype="multipart/form-data">
          <input type="file" name="image">
          <input type="submit" value="Upload">
    </form>
</body>
</html>
```

File Uploads (II): serve the HTML form

The program reads this HTML from `upload.html` and displays it to clients that access the web server on port 8080.

```
package main

import "http"
import "template"

var uploadTemplate = template.MustParseFile("upload.html", nil)

func upload(w http.ResponseWriter, r *http.Request) {
    uploadTemplate.Execute(w, nil)
}

func main() {
    http.HandleFunc("/", upload)
    http.ListenAndServe(":8080", nil)
}
```

File Uploads (III): handle the uploaded file

```
func upload(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        uploadTemplate.Execute(w, nil)
        return
    }
    f, _, err := r.FormFile("image")
    if err != nil {
        http.Error(w, err.String(), 500)
        return
    }
    defer f.Close()
    t, err := ioutil.TempFile(".", "image-")
    if err != nil {
        http.Error(w, err.String(), 500)
        return
    }
    defer t.Close()
    if _, err := io.Copy(t, f); err != nil {
        http.Error(w, err.String(), 500)
        return
    }
    http.Redirect(w, r, "/view?id="+t.Name()[6:], 302)
}
```

An aside: improved error handling (I)

```
var errorTemplate = template.MustParseFile("error.html", nil)

func errorHandler(fn http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if e, ok := recover().(os.Error); ok {
                w.WriteHeader(500)
                errorTemplate.Execute(w, e)
            }
        }()
        fn(w, r)
    }
}

func main() {
    http.HandleFunc("/", errorHandler(upload))
    http.ListenAndServe(":8080", nil)
}
```

An aside: improved error handling (II)

```
func check(err os.Error) { if err != nil { panic(err) } }

func upload(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        uploadTemplate.Execute(w, nil)
        return
    }
    f, _, err := r.FormFile("image")
    check(err)
    defer f.Close()

    t, err := ioutil.TempFile(".", "image-")
    check(err)
    defer t.Close()

    _, err := io.Copy(t, f)
    check(err)
    http.Redirect(w, r, "/view?id="+t.Name()[6:], 302)
}
```

Serve the image

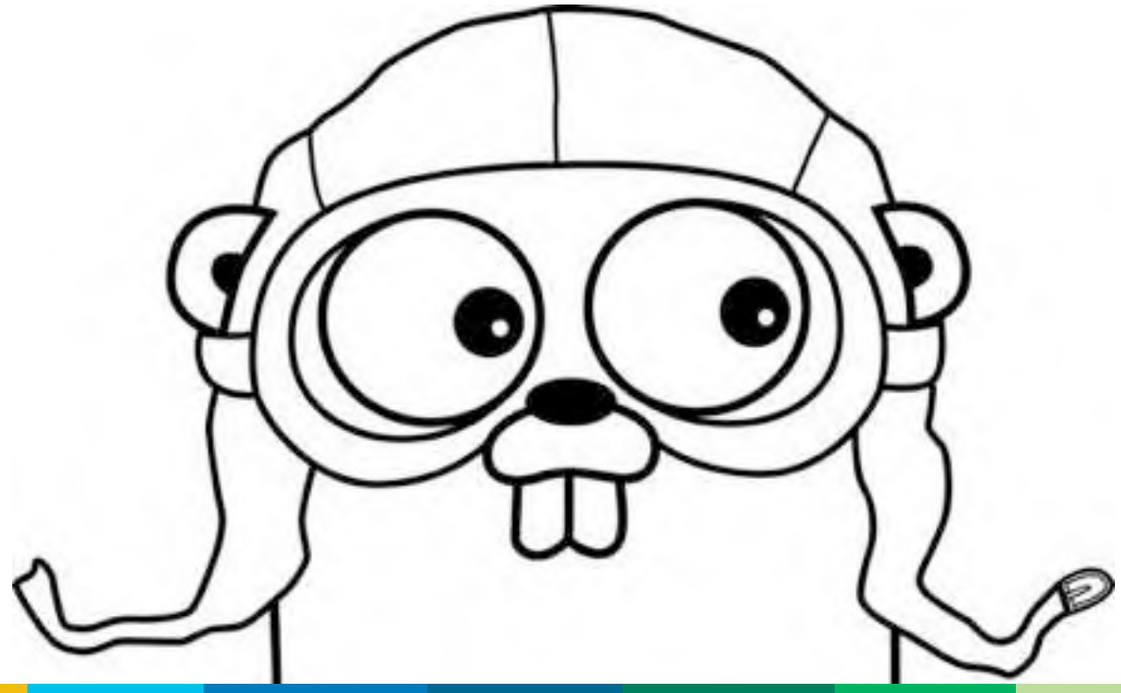
A new handler, `view`, uses the `ServeFile` helper function to read the image from disk and send it as the HTTP response.

```
func view(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "image")
    http.ServeFile(w, r, "image-"+r.FormValue("id"))
}
```

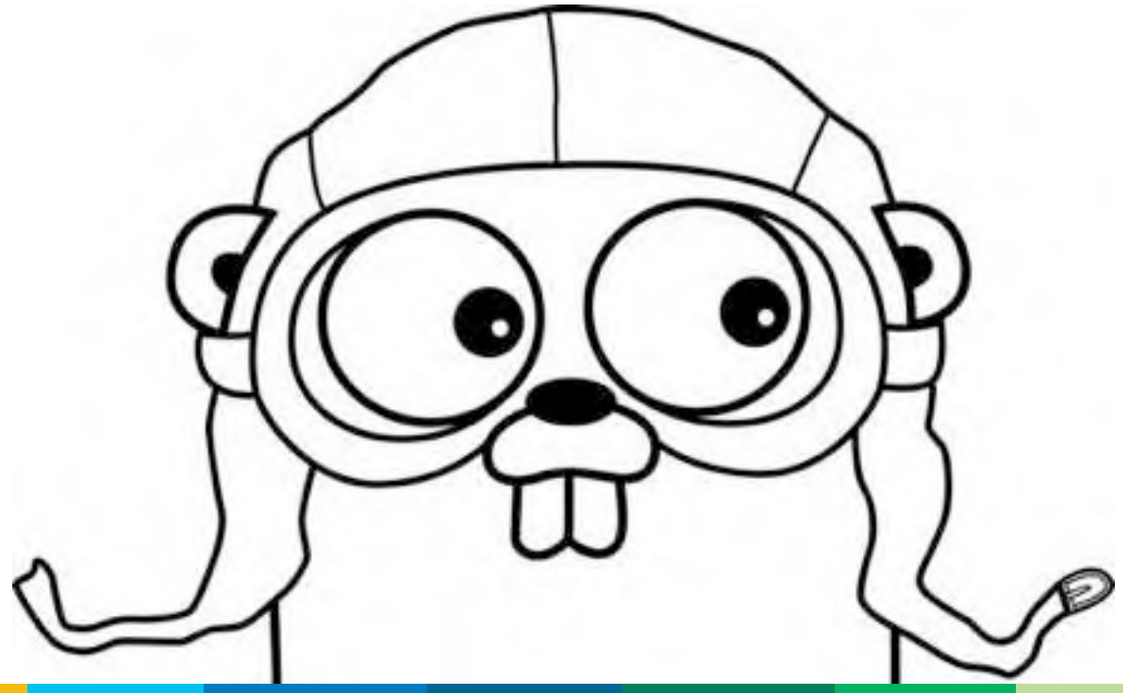
The `view` handler serves requests to `/view`, while the `upload` handler continues to serve all other requests.

```
func main() {
    http.HandleFunc("/", errorHandler(upload))
    http.HandleFunc("/view", errorHandler(view))
    http.ListenAndServe(":8080", nil)
}
```

Demo



Moustaches



Drawing a Moustache: the `raster` library

The `freetype-go` library is a native Go implementation of the FreeType font rendering library. Its `raster` package provides some anti-aliased curve rasterization primitives we can use to draw a moustache.

This external package can be installed using the `goinstall` command:

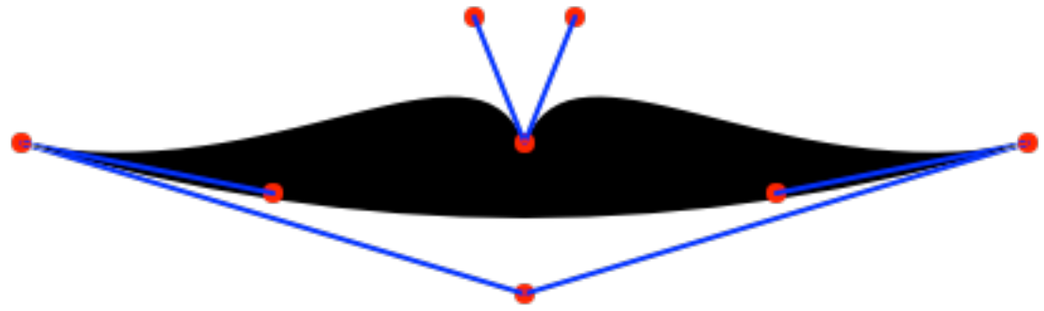
```
goinstall freetype-go.googlecode.com/hg/freetype/raster
```

We then import the `raster` package by the same path:

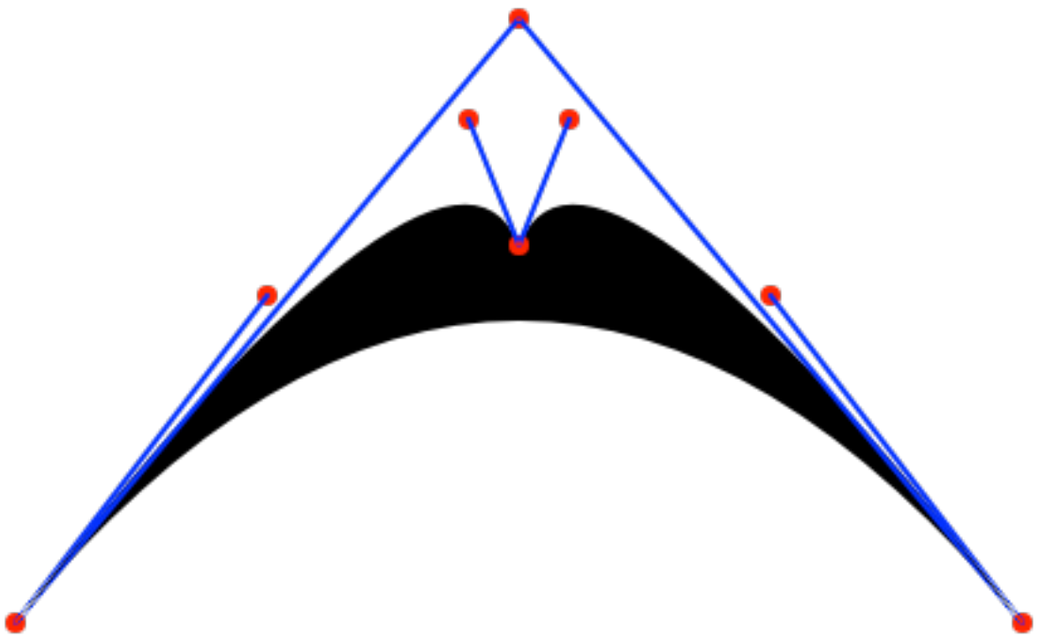
```
import "freetype-go.googlecode.com/hg/freetype/raster"
```

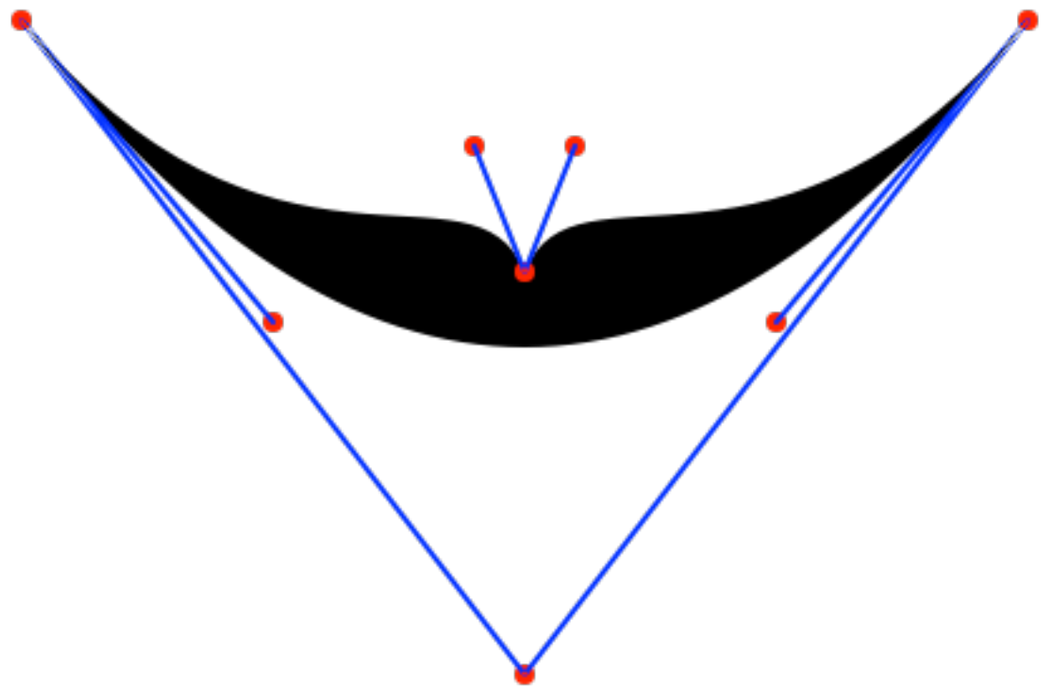
Drawing a Moustache: the rakish appearance

Our moustache is made from three Bézier curves.



We can alter the “droop factor” by moving the control points.





Drawing a Moustache: the `moustache` function (I)

```
func moustache(m image.Image, x, y, size, droopFactor int) image.Image {
    mrgba := rgba(m) // Create specialized RGBA image from m.

    p := raster.NewRGBAPainter(mrgba)
    p.SetColor(image.RGBAColor{0, 0, 0, 255})

    w, h := m.Bounds().Dx(), m.Bounds().Dy()
    r := raster.NewRasterizer(w, h)
    var (
        mag   = raster.Fix32((10 + size) << 8)
        width = pt(20, 0).Mul(mag)
        mid   = pt(x, y)
        droop = pt(0, droopFactor).Mul(mag)
        left  = mid.Sub(width).Add(droop)
        right = mid.Add(width).Add(droop)
        bow   = pt(0, 5).Mul(mag).Sub(droop)
        curlx = pt(10, 0).Mul(mag)
        curly = pt(0, 2).Mul(mag)
        risex = pt(2, 0).Mul(mag)
        risey = pt(0, 5).Mul(mag)
    )
    // continued on next slide...
```

Drawing a Moustache: the `moustache` function (II)

```
    r.Start(left)
    r.Add3(
        mid.Sub(curlx).Add(curly),
        mid.Sub(risex).Sub(risey),
        mid,
    )
    r.Add3(
        mid.Add(risex).Sub(risey),
        mid.Add(curlx).Add(curly),
        right,
    )
    r.Add2(
        mid.Add(bow),
        left,
    )
    r.Rasterize(p)

    return mrgba
}
```

Drawing a Moustache: converting image types

An `image.Image` is an interface type that can hold arbitrary image types (RGBA, greyscale, fixed-palette, and so on).

The `raster` library only handles a specific subset of image types. Our moustache routine draws on an `*image.RGBA`. A user might upload any kind of image, so we may need to convert it before we can draw on it.

```
func moustache(m image.Image, ...) image.Image {
    mrgba := rgba(m)
    p := raster.NewRGBAPainter(mrgba)
    // ...
```

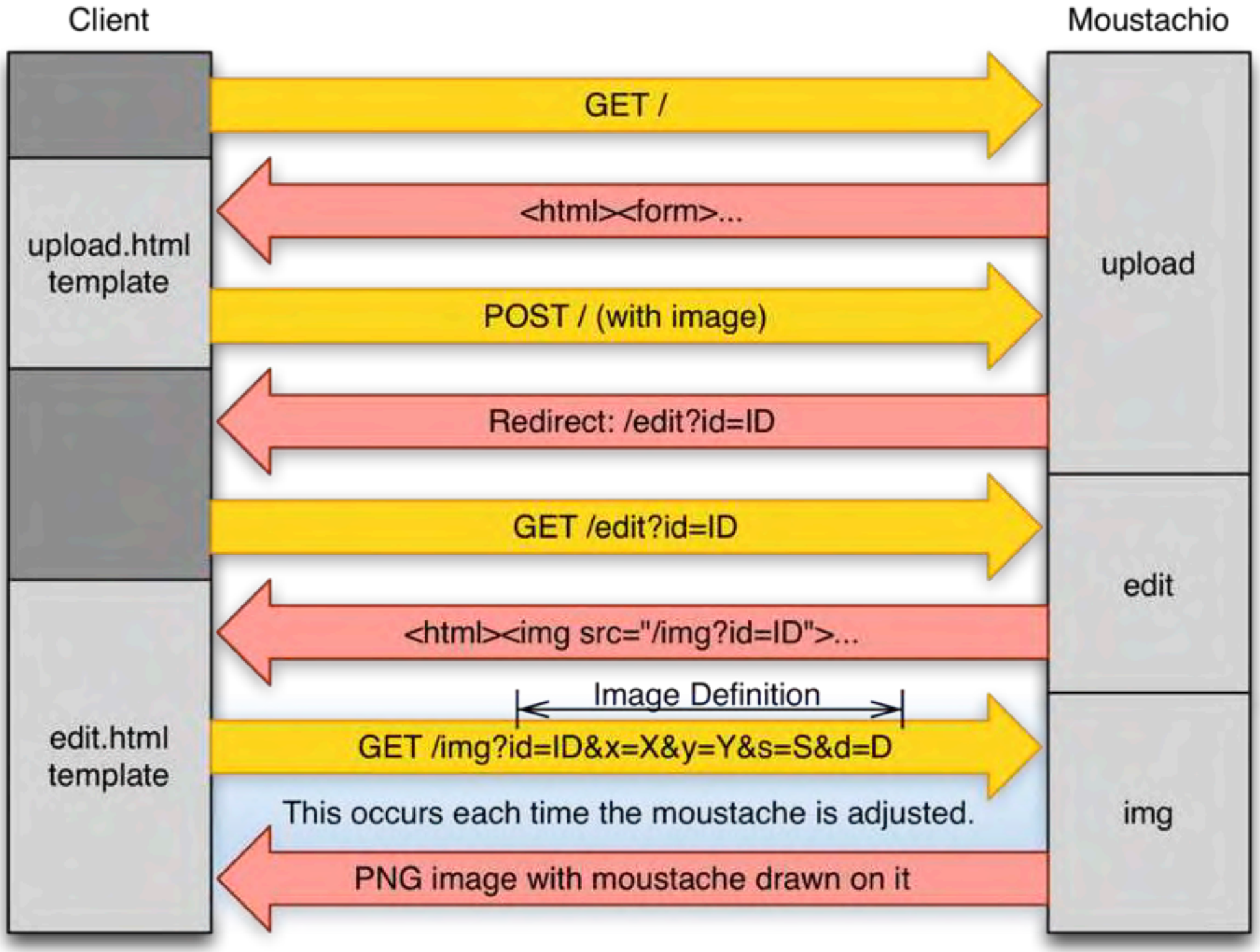
To do this, we use the `rgba` function. It takes an `image.Image` and returns an `*image.RGBA`. But how does it work?

Drawing a Moustache: converting image types

The `rgba` function performs a type assertion to see if the underlying concrete type is an `*image.RGBA`. If so, it simply returns the existing image as its concrete type. Otherwise, it creates a new `*image.RGBA` and copies the image data to it.

```
func rgba(m image.Image) *image.RGBA {
    // Fast path: if m is already an RGBA, just return it.
    if r, ok := m.(*image.RGBA); ok {
        return r
    }
    // Create a new image and draw m into it.
    b := m.Bounds()
    r := image.NewRGBA(b.Dx(), b.Dy())
    draw.Draw(r, b, m, image.ZP)
    return r
}
```

Data flow: uploading and drawing on images



Drawing a Moustache: the user interface

The `edit` handler serves an HTML page containing some JavaScript code that provides a user interface for adjusting the parameters of the moustache.

On the client side, those parameters are translated into a request to `/img` with an appropriate query string.

```
var editTemplate = template.MustParseFile("edit.html", nil)

func edit(w http.ResponseWriter, r *http.Request) {
    editTemplate.Execute(w, r.FormValue("id"))
}
```

The image ID is passed through to the template so the JavaScript client knows which image to use.

Drawing a Moustache: decoding and encoding (I)

The `img` handler reads and decodes the specified image, draws a moustache on it, and serves it as a JPEG image.

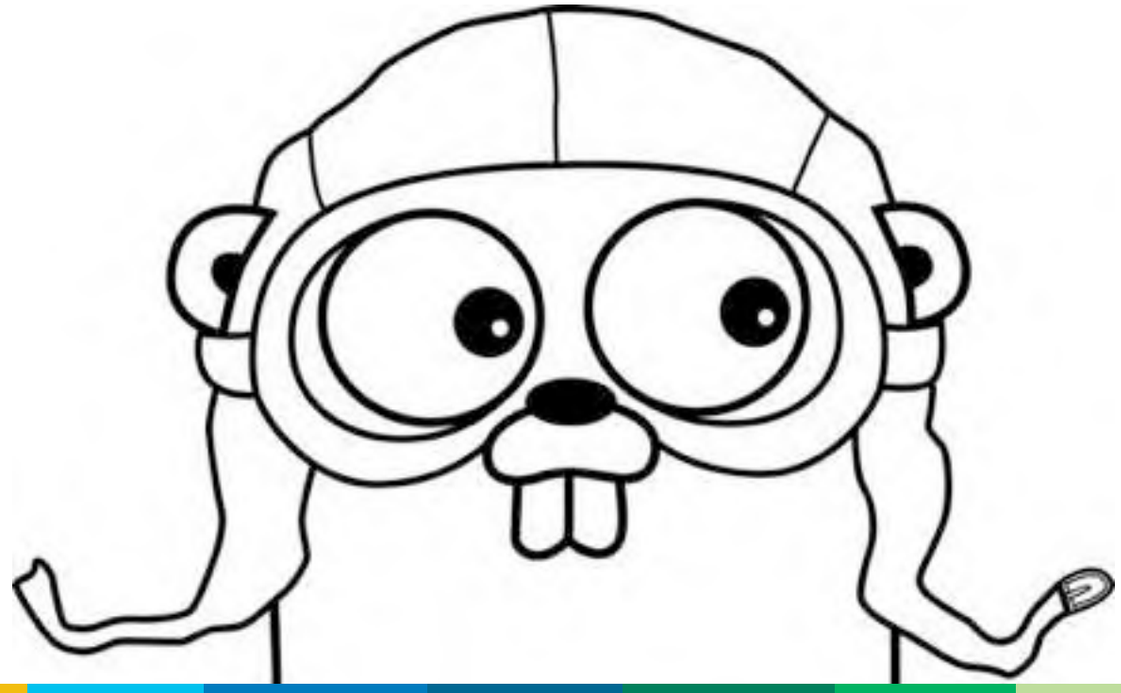
```
func img(w http.ResponseWriter, r *http.Request) {
    f, err := os.Open("image-"+r.FormValue("id"))
    check(err)
    m, _, err := image.Decode(f)
    check(err)

    x, _ := strconv.Atoi(r.FormValue("x"))
    y, _ := strconv.Atoi(r.FormValue("y"))
    s, _ := strconv.Atoi(r.FormValue("s"))
    d, _ := strconv.Atoi(r.FormValue("d"))
    m = moustache(m, x, y, s, d)

    w.Header().Set("Content-type", "image/jpeg")
    jpeg.Encode(w, m, nil) // Default JPEG options.
}
```

Drawing a Moustache: decoding and encoding (I)

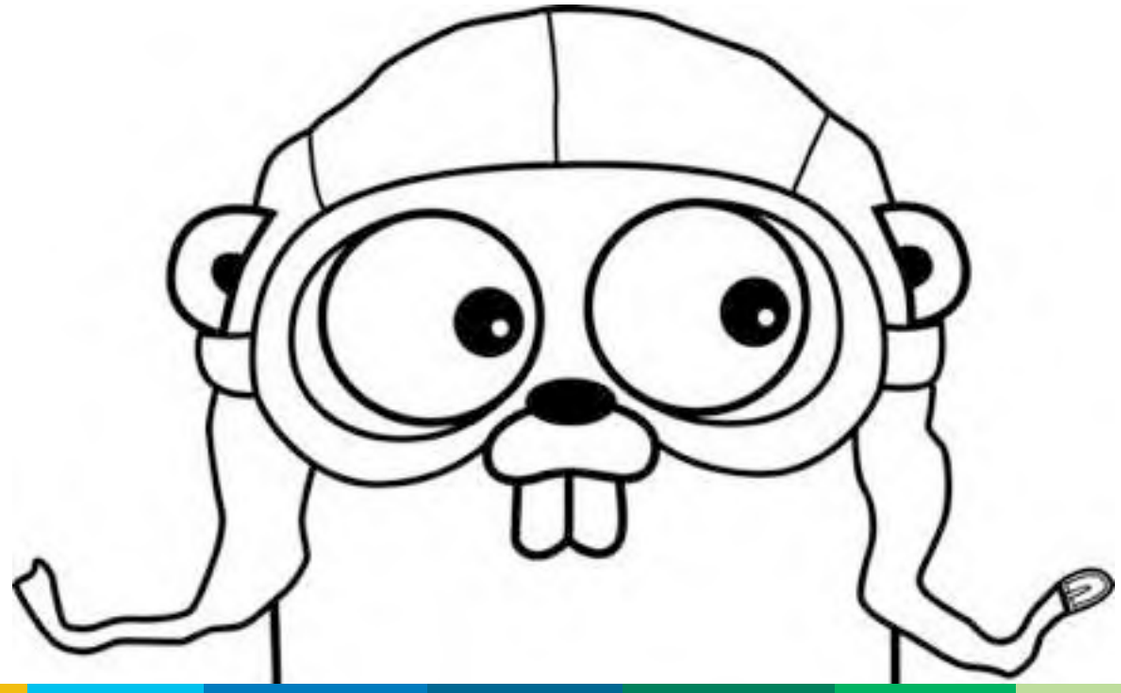
The `img` handler reads and decodes the specified image, draws a moustache on it, and serves it as a JPEG image.

```
func img(w http.ResponseWriter, r *http.Request) {
    f, err := os.Open("image-"+r.FormValue("id"))
    check(err)
    m, _, err := image.Decode(f)
    check(err)

    v := func(n string) int { // helper closure
        i, _ := strconv.Atoi(r.FormValue(n))
        return i
    }
    m = moustache(m, v("x"), v("y"), v("s"), v("d"))

    w.Header().Set("Content-type", "image/jpeg")
    jpeg.Encode(w, m, nil) // Default JPEG options.
}
```

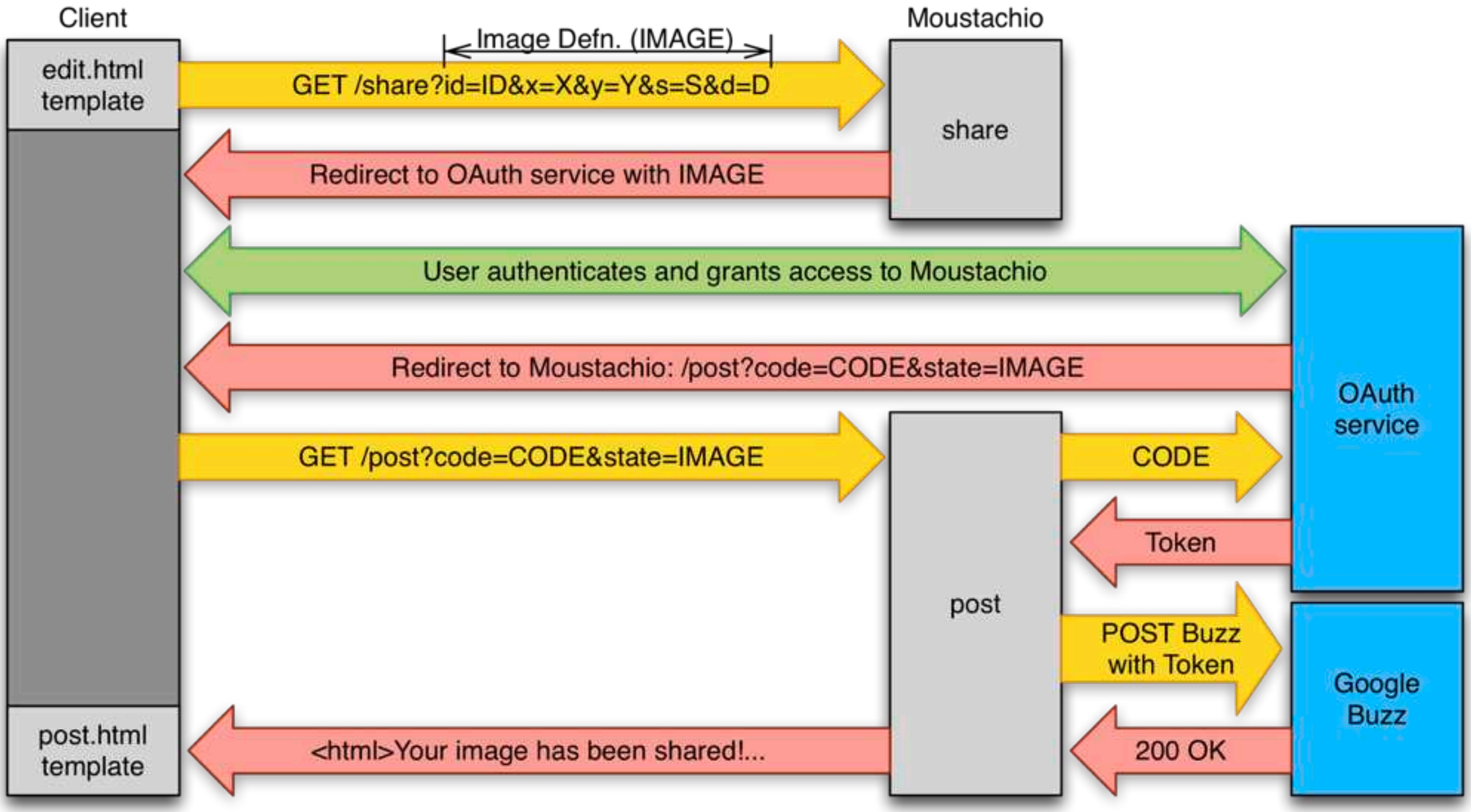
Demo



Sharing



Data flow: authenticating with and posting to Buzz



Sharing the image: the goauth2 package

To share the image via Google Buzz we must authenticate with Google's servers using the OAuth2 protocol.

We will use another external library, goauth2, also available via goinstall:

```
goinstall goauth2.googlecode.com/hg/oauth
```

Again, we import the package using the same path:

```
import "goauth2.googlecode.com/hg/oauth"
```

Sharing the image: authenticating

The OAuth client needs some configuration:

```
var config = &oauth.Config{
    ClientId:     OAUTH_CLIENT_ID,
    ClientSecret: OAUTH_CLIENT_SECRET,
    Scope:        "https://www.googleapis.com/auth/buzz",
    AuthURL:      "https://accounts.google.com/o/oauth2/auth",
    TokenURL:     "https://accounts.google.com/o/oauth2/token",
    RedirectURL:  "http://moustachio/post",
}
```

The `share` handler redirects the user to Google's OAuth service:

```
func share(w http.ResponseWriter, r *http.Request) {
    url := config.AuthCodeURL(r.URL.RawQuery)
    http.Redirect(w, r, url, 302)
}
```

Sharing the image: posting to Buzz

The `post` handler accepts the authentication `code` and `image` state from the OAuth service, exchanges the `code` for an OAuth authentication token, and posts to Buzz:

```
func post(w http.ResponseWriter, r *http.Request) {
    t := &oauth.Transport{Config: config}
    code := r.FormValue("code")
    _, err := t.Exchange(code)
    check(err)

    image := r.FormValue("state")
    err = postPhoto(t.Client(), "http://moustachio/img?"+image)
    check(err)

    postTemplate.Execute(w, url)
}
```

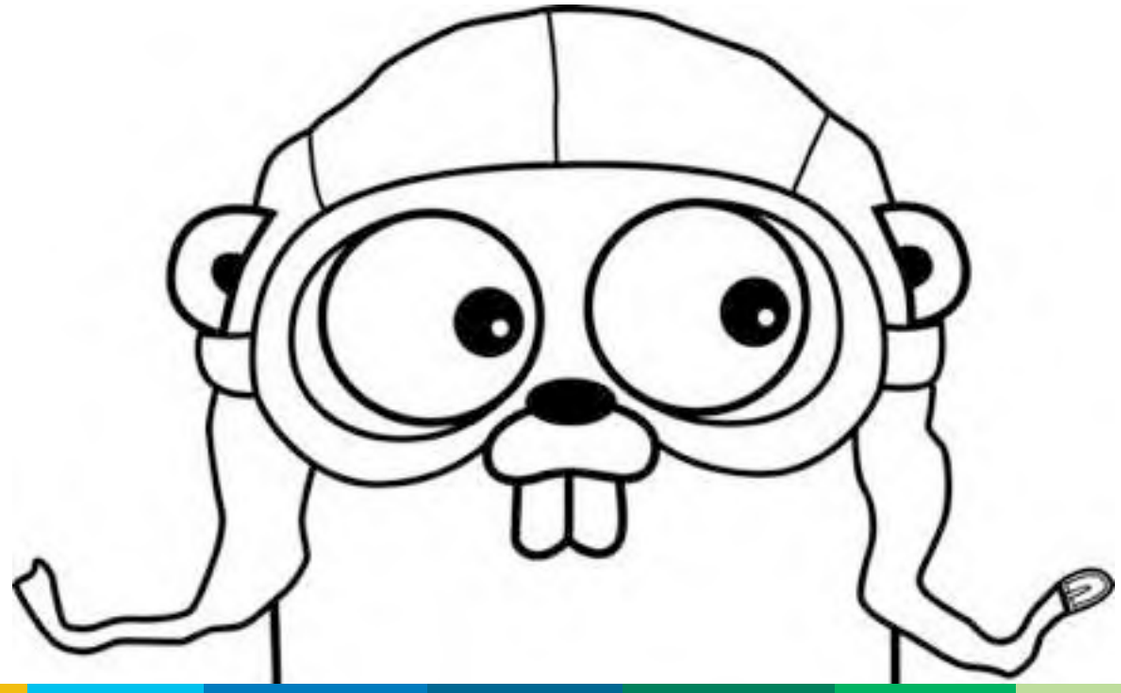
Sharing the image: posting to Buzz

The `postPhoto` function forms a JSON-encoded request and makes an HTTP `POST` to the Buzz API. It does this through the provided `*http.Client` to authenticate the request.

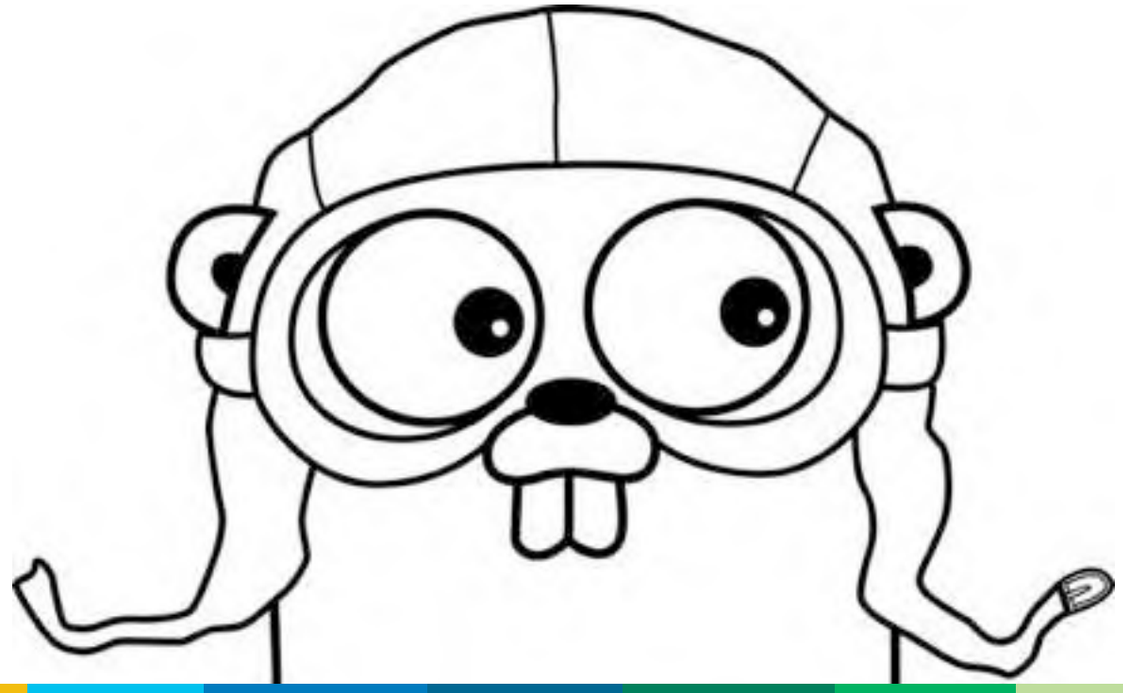
```
func postPhoto(client *http.Client, photoURL string) os.Error {
    // omitted: url is the URL of the Buzz API
    //          req is an API request encoded as JSON
    resp, err := client.Post(url, "application/json", req)
    if err != nil {
        return err
    }
    if resp.StatusCode != 200 {
        return os.NewError("invalid post " + resp.Status)
    }
    return nil
}
```

A general Go client for Google APIs is under development. It should simplify much of this code.

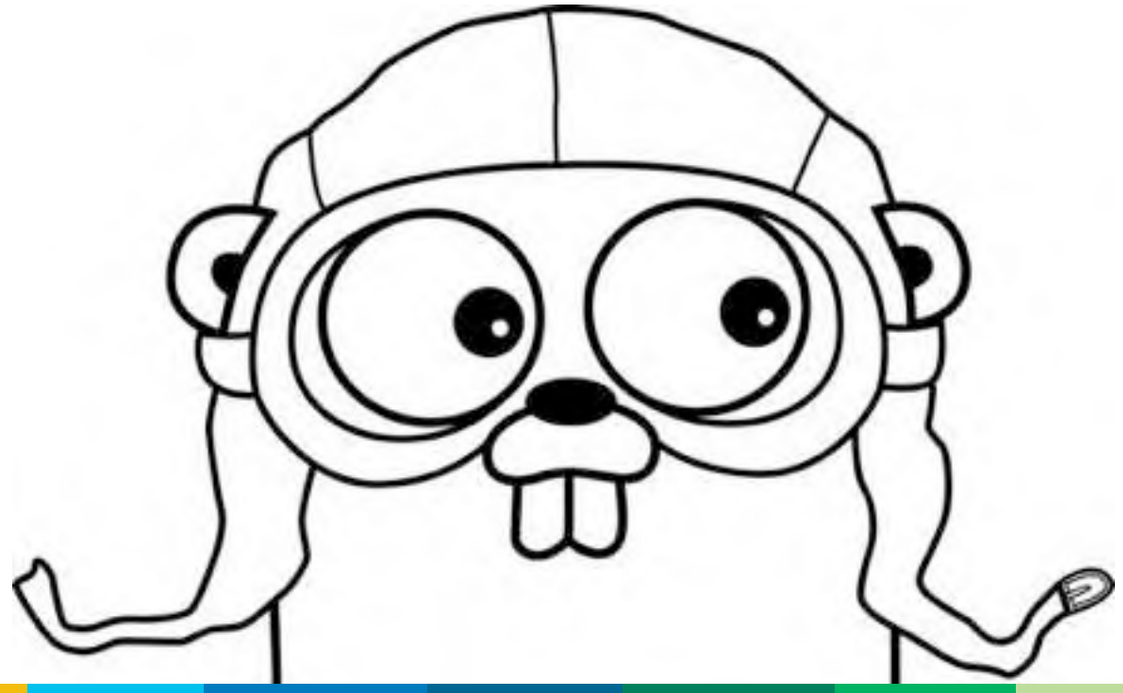
Demo



Going to production



Go on App Engine



Making Moustachio work on App Engine

App Engine provides the `main` package and HTTP listener, so change package name, move setup into an `init` function and drop call to `http.ListenAndServe`. Also add a couple of imports.

```
package moustachio

import (
    "appengine"
    "appengine/datastore"
)

func init() {
    http.HandleFunc("/", errorHandler(upload))
    http.HandleFunc("/edit", errorHandler(edit))
    http.HandleFunc("/img", errorHandler(img))
    http.HandleFunc("/share", errorHandler(share))
    http.HandleFunc("/post", errorHandler(post))
}
```

Store the data in App Engine data store

First we define a simple data structure to hold the image data.

```
type Image struct {
    Data []byte
}
```

We could store the information in the App Engine blob store, but the data store makes it easier to add other information to the image as the program evolves.

Next, we modify the upload and image fetch code to store the data in the data store, under a unique key.

Creating the key

The key is created by hashing the image data.
SHA-1 is overkill but easy; just a few bytes of hash are good enough for our purpose.

```
// keyOf returns (part of) the SHA-1 hash of the data,
// as a hex string.
func keyOf(data []byte) string {
    sha := sha1.New()
    sha.Write(data)
    return fmt.Sprintf("%x", string(sha.Sum())[0:16])
}
```

Store uploaded image in App Engine data store

```
func upload(w http.ResponseWriter, r *http.Request) {
    // ... same as before until we have the data in hand...
    // Grab the image data
    buf := new(bytes.Buffer)
    _, err = io.Copy(buf, f)
    check(err)

    // Create an App Engine context for the client's request.
    c := appengine.NewContext(r)

    // Save the image under a unique key, a hash of the image.
    key := datastore.NewKey("Image", keyOf(buf.Bytes()), 0, nil)
    _, err = datastore.Put(c, key, &Image{buf.Bytes()})
    check(err)

    // Redirect to /edit using the key.
    http.Redirect(w, r, "/edit?id="+key.StringID(),
                  http.StatusFound)
}
```

Grab the data from the data store

When a request comes in, use the key to recover the image data from the data store.

```
func img(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)

    key := datastore.NewKey("Image", r.FormValue("id"), 0, nil)
    im := new(Image)
    err := datastore.Get(c, key, im)
    check(err)

    i, _, err := image.Decode(bytes.NewBuffer(im.Data))
    check(err)

    // ... The rest is as before
}
```

Use the URL Fetch API to make HTTP requests

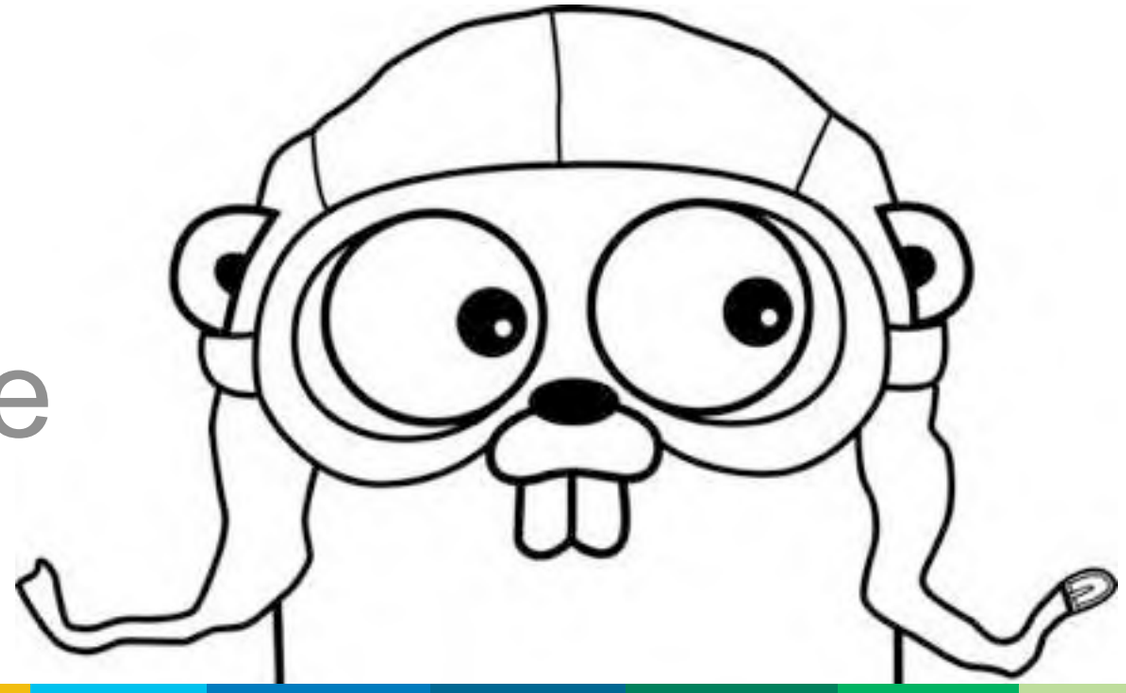
The `appengine/urlfetch` package's `Transport` type makes HTTP requests through App Engine's infrastructure.

The `oauth.Transport` type permits the use of a custom HTTP transport.

If we put an instance of `urlfetch.Transport` inside our `oauth.Transport`, the rest of our code works as is.

```
t := &oauth.Transport{
    Config: conf,
    Transport: &urlfetch.Transport{
        Context: appengine.NewContext(r),
    },
}
```

Deploy to App Engine



Configure the app

Create a simple `app.yaml` file that states we are using the Go runtime at version 1 and that our Go program should serve all requests.

```
application: your-app-id
version: 1
runtime: go
api_version: 1

handlers:
- url: /.*
  script: _go_app
```

Assembling the app

So that App Engine can build our app automatically, we put our Go source files in a `moustachio` directory and copy the `freetype-go` and `goauth2` packages to our application root.

The application file hierarchy looks like this:

```
app.yaml
edit.html
error.html
post.html
upload.html
moustachio/draw.go
moustachio/http.go
freetype-go.googlecode.com/hg/freetype/raster/...
goauth2.googlecode.com/hg/oauth/...
```

Testing and deploying the app

To test the application, we use the same development application server used by the App Engine Python SDK:

```
dev_appserver.py /path/to/app
```

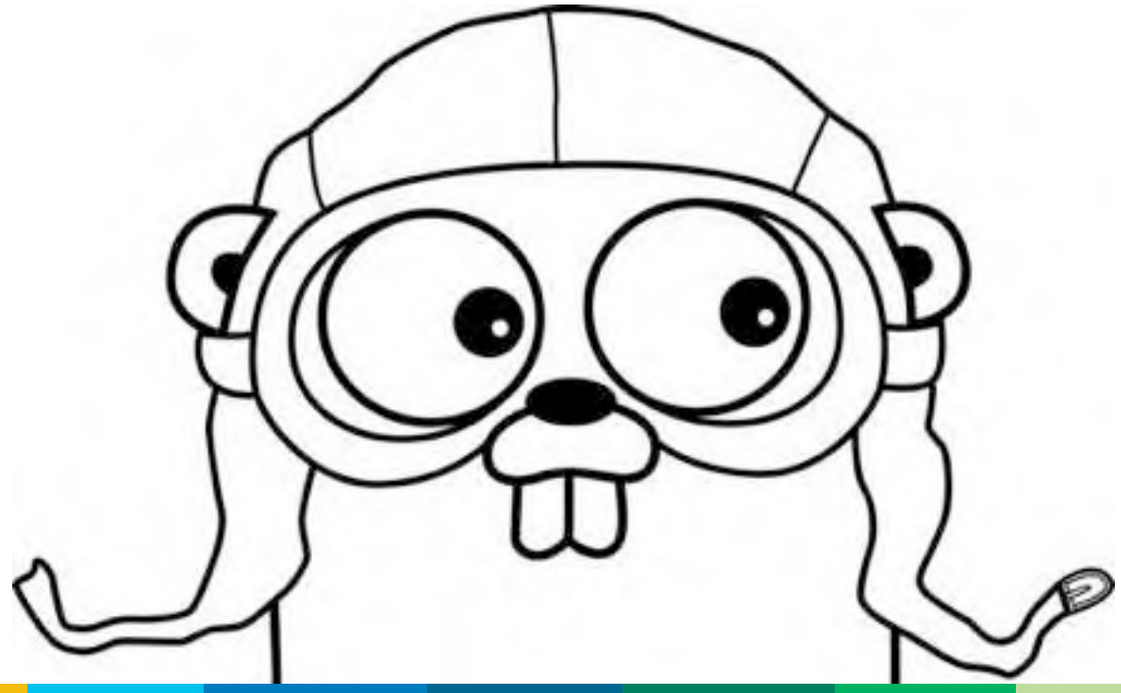
To deploy the application we must first create an App Engine app through the admin console in the usual way, and then deploy moustachio with `appcfg.py`:

```
appcfg.py update /path/to/app
```

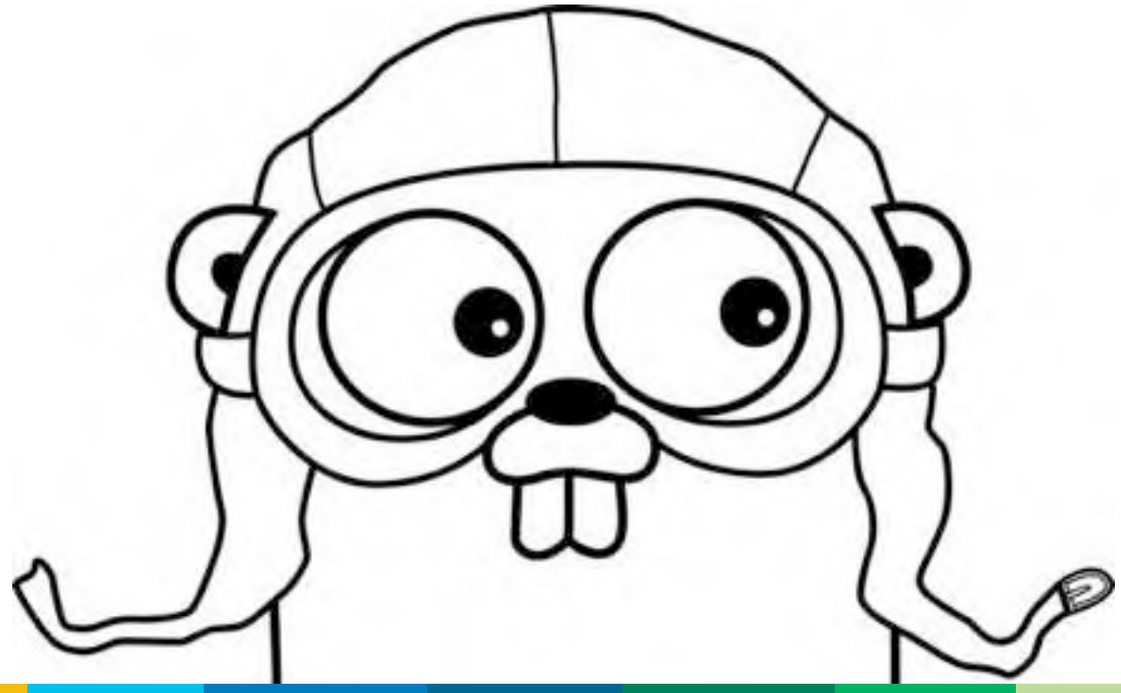
See it running live, right away, at your app's URL:

```
http://your-app-id.appspot.com
```

Demo



Availability?

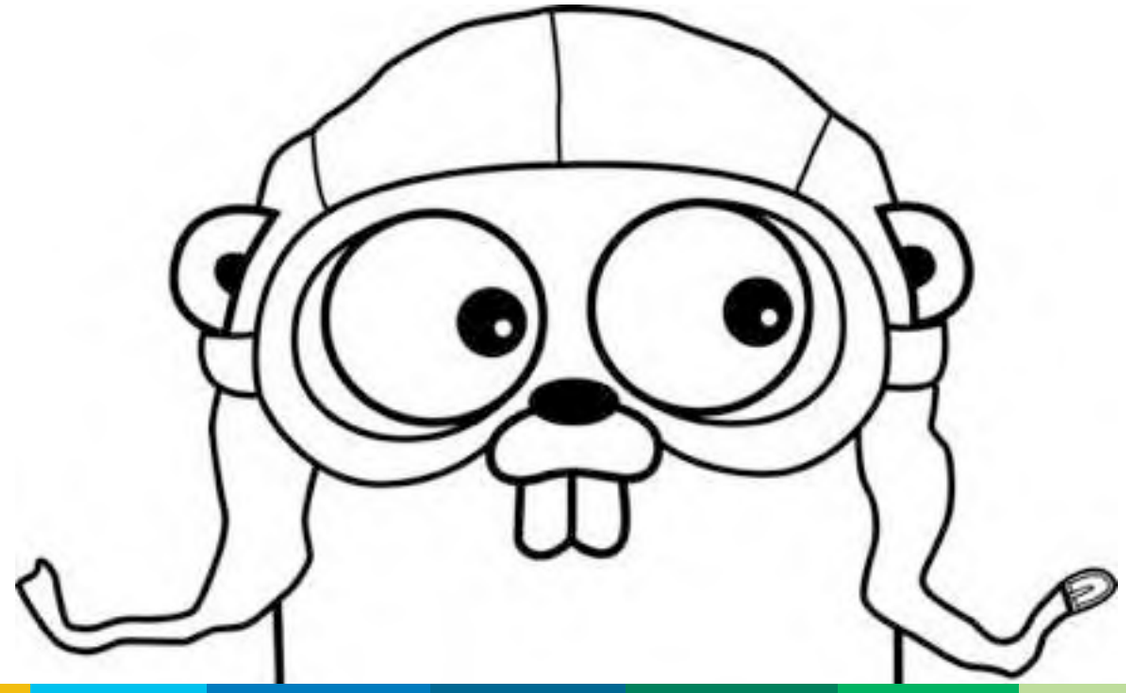


Go on App Engine: Launch

- The Go App Engine runtime is an experimental feature.
- The SDK is publicly available for download.
- Production deployment available to Trusted Testers soon.
- Most of the App Engine APIs are supported:
  Datastore, Memcache, URL Fetch, Users, Mail, Blobstore,
  Task Queues, and more coming soon.
- For full details (and to sign up as a Trusted Tester)
  see the announcement on the Go blog:
  - `http://blog.golang.org/`

Credit to David Symonds, Nigel Tao, Andrew Gerrand,
and the rest of the Google App Engine and Go Teams.

Conclusion



Go is a great back-end language for App Engine

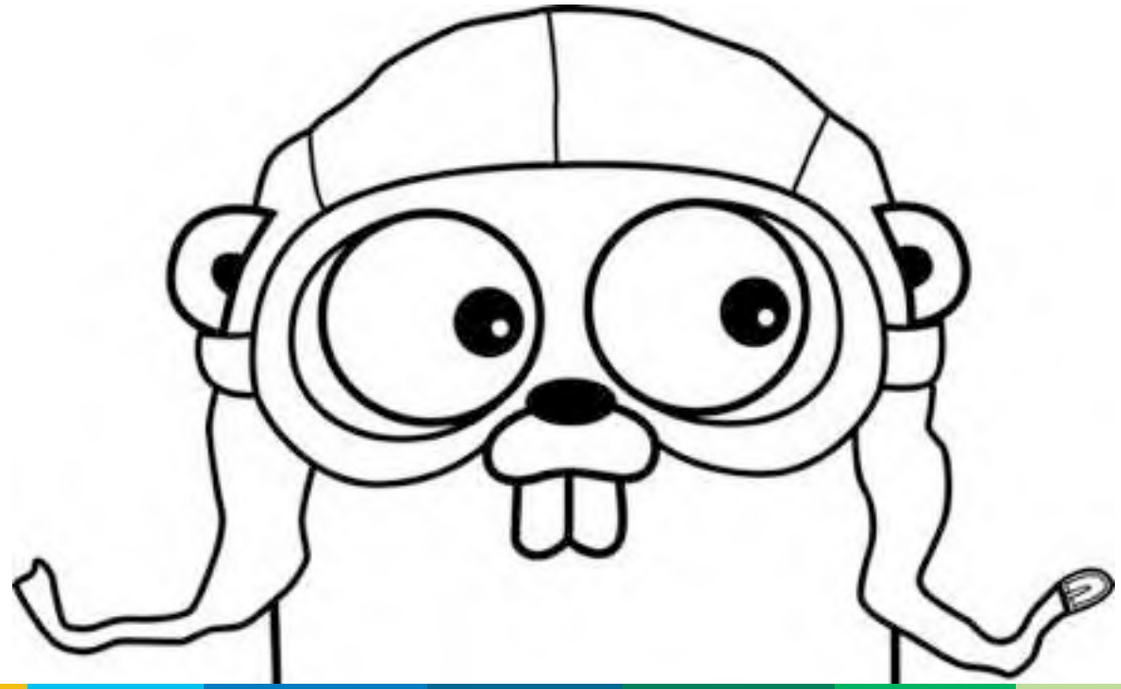
- A high performance language for App Engine
  - easy and productive to use.
  - scales well.
  - good (and ever-improving) library support.
  - efficient – a true compiled language.

- Start using the Go App Engine SDK today:
  - `http://code.google.com/appengine/downloads.html`
- Full source code for SDK and this example available at:
  - `http://code.google.com/p/appengine-go/`
- Try Moustachio on the public App Engine service:
  - `http://moustach-io.appspot.com`

More about Go

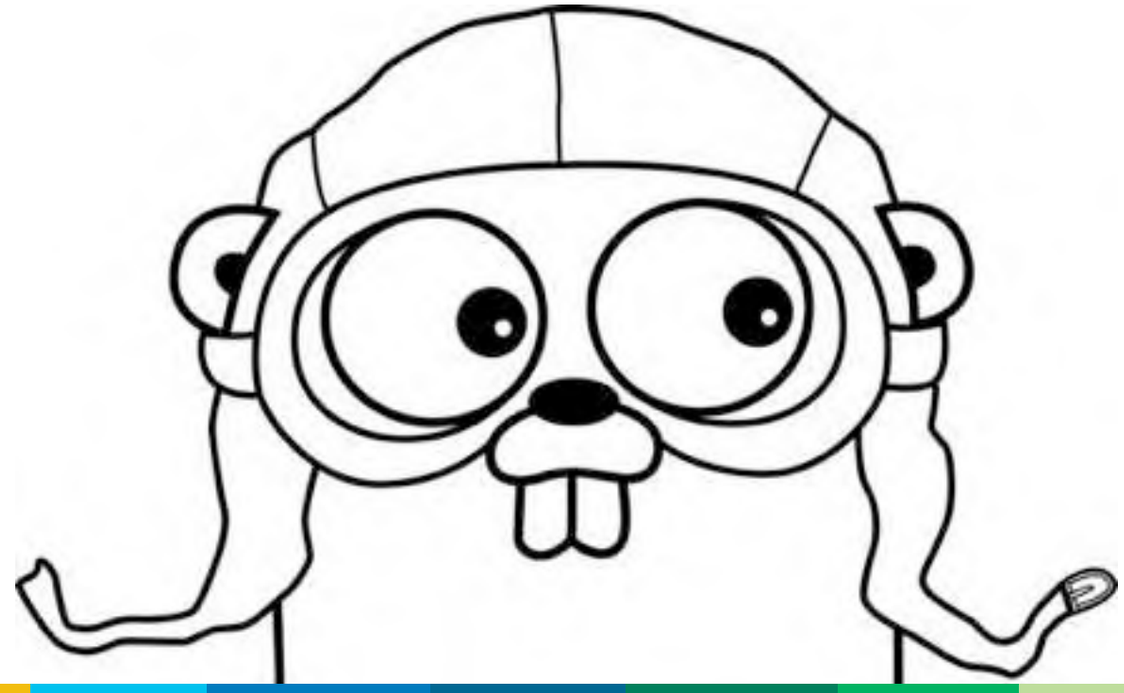
- The Go web site `golang.org` is a valuable resource:
  - tutorials, code walks, videos,
  - package and command references,
  - language specification. (Readable, and worth reading!)
- The Go blog
  - `http://blog.golang.org`
- Go on App Engine
  - `http://code.google.com/appengine/docs/go/`
- Go Office Hours at I/O
  - 12pm to 3pm today, and with the App Engine team tomorrow

Questions?

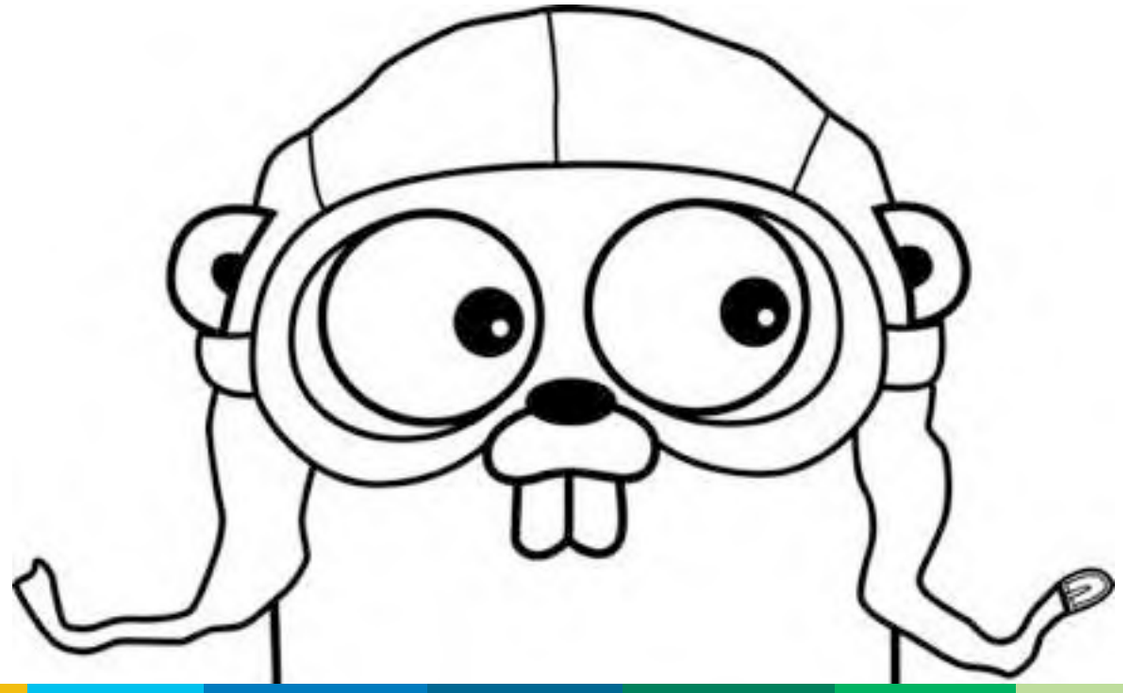


Feedback

http://goo.gl/U5rYj
#io2011 #DevTools

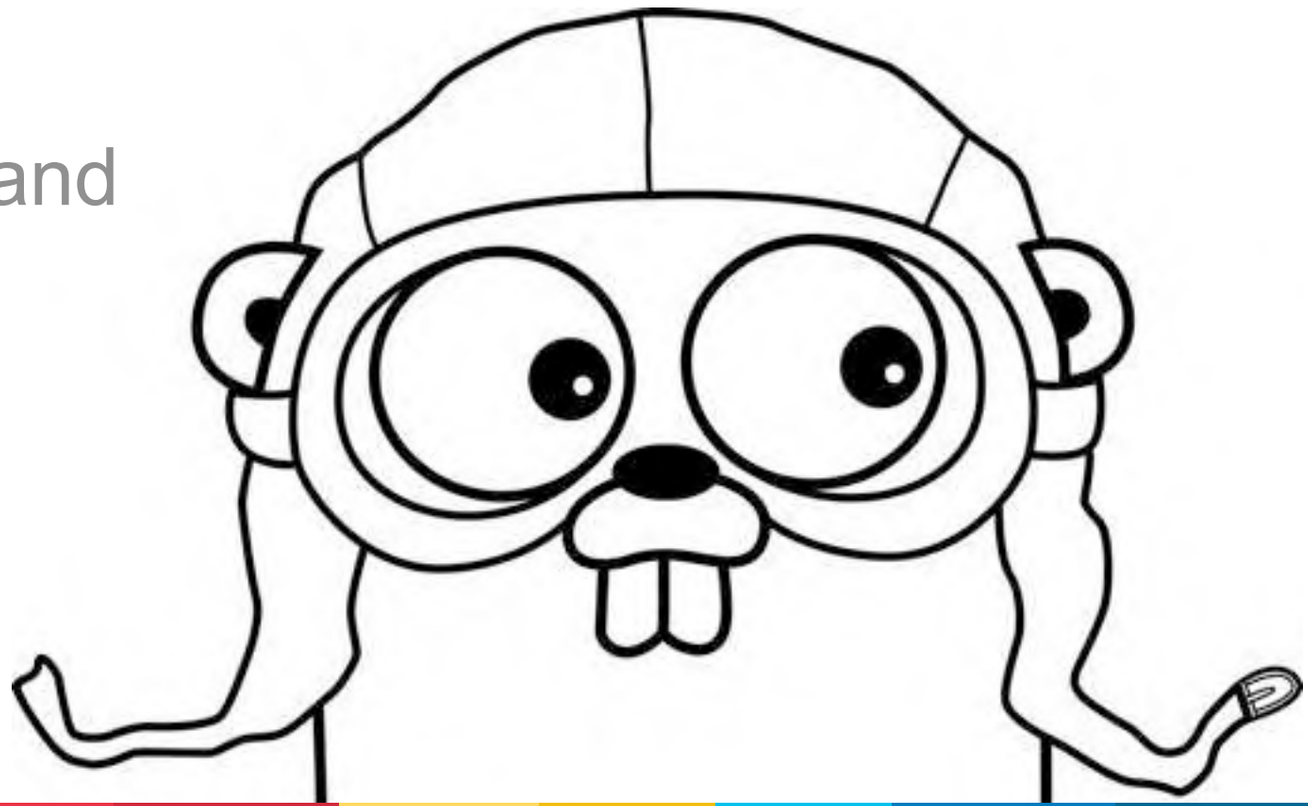


Gophers!



Writing Web Apps in Go

Andrew Gerrand
Rob Pike
May 10, 2011



Google I/O 11


Google™
11
I/O