# How we use Go to build APIs

## PubNative best practices and patterns

Kostiantyn Stepaniuk
PubNative GmbH

## 1/17 About PubNative

- Adtech company which connects publishers and advertisers through its technology
- First line of the API code was written in **December 2013** in **Ruby**
- In **November 2014** we migrated to **Go**
- API and supportive utilities have **200K** lines of **Go** code
- We process **12M req/min**
- Response time below **100ms**
- We are connected to **197** partners with who we are doing real-time auction
- At a peak, we send **12M*197=2.364B** outbound req/min

## 2/17 What challenges we have

- **Competitive**: cost-efficient and high-performing
- **Resilient**: survives the outage of any used service, DB, queue, cache, etc.
- **Quick TTM**: 1-2 products/year
- **Productivity**: 3043 PRs closed since 2013 or 2 PRs/day
- **Efficiency**: product/tech task split 85/15, developing by 4 people (of 14 engineers)

## 3/17 How we do it

Make it simple
Make it boring
Focus on what matters

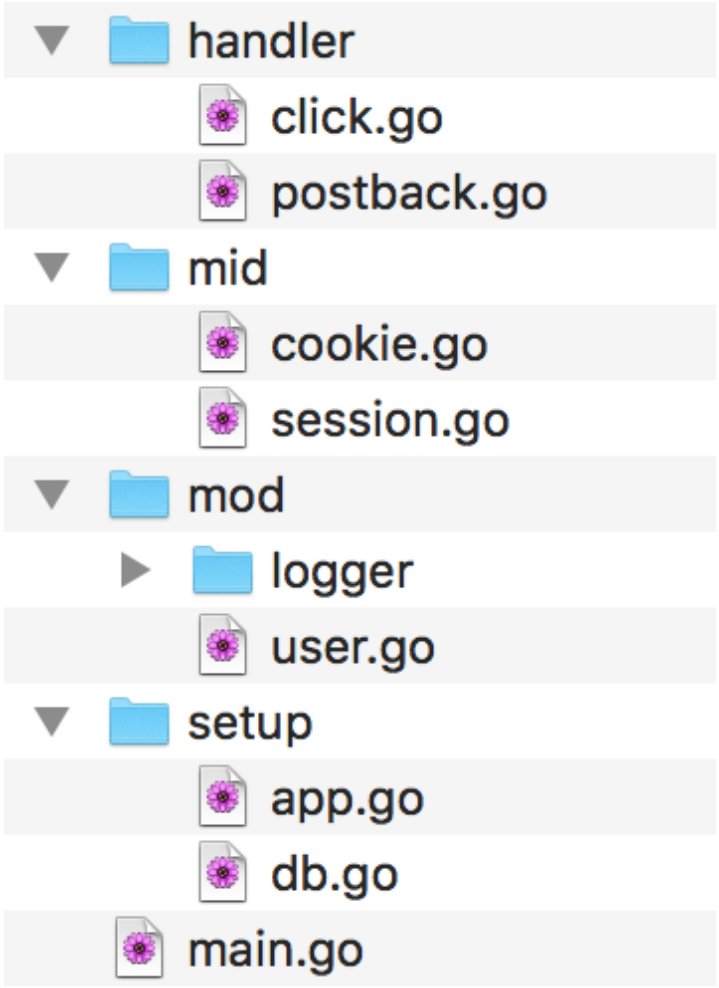
# 4/17 Learn techniques

Painted in 1897 by 15-year-old boy

## 5/17 And then improve

Painted in 1955 by our 74-year-old boy :)

# 6/17 Use the same structure



# 7/17 Full control over HTTP protocol

```
http.HandleFunc("/handle/b", func(http.ResponseWriter, *http.Request) {})
http.HandleFunc("/handle/a", func(http.ResponseWriter, *http.Request) {})

http.HandleFunc("/basic_auth", func(w http.ResponseWriter, r *http.Request) {
    usr, pwd, _ := r.BasicAuth()
    if usr != "user" || pwd != "password" {
        w.Header().Set("WWW-Authenticate", `Basic realm="Application"`)
        w.WriteHeader(401)
    }
})
```

- No powerful URL routers like gorilla/mux (https://github.com/gorilla/mux) because of performance penalty and decreasing clarity. Go provides all the necessary wrappers
- Don't carry parameters in the path, use a query string or a request payload
- We need to know what happens under the hood to optimize

# 8/17 Use explicit types, avoid interfaces

```
type App struct {
    Redis *redis.Client       // explicit name
    HB    *honeybadger.Client //commonly used abbreviation
}
```

- We solve specific problems with concrete tools
- Clear understanding what is used and how
- Renaming can be done in one command in your favorite editor

```
type ObjectA struct{} // Do we really need to have one interface?
type ObjectB struct{} // Or we can keep them separately
func (o ObjectA) Run(task int) bool {}
func (o ObjectB) Do(task int) bool {}
```

- Got a challenge, try to solve it in the best way possible and then think of generalizing, not another way around

# 9/17 One way of doing things

```
type A struct {
    B B
}
type B struct {
    C C
}
type C struct {
    D string
}

a := A{}
a.B.C.D // access D through chain

a.GetD() // NO helper methods

a.C.D // NO aliases
```

It also applies how to create pools, integrates 3rd parties or writes tests, etc.

# 10/17 Own pool of workers instead of what libraries provide

```
func main() {
    pool := 10
    ch := make(chan string, pool+10) // how many messages keep in a memory before taking an action
    for i := 0; i < pool; i++ {
        conn, err := redis.Dial("tcp", ":6379") // connection per worker
        _ = err // handle error
        go func() {
            conn.Do("HSET", "hash", <-ch, 1)
        }()
    }
}
```

- Know how many connections are used
- No hidden bottlenecks, e.g., locks are used to manage the pool
- Easy to profile the pool

# 11/17 Reserve a worker for a task

```
func main() {
    jobs := make(chan chan int) // first chan to reserve a worker, second one to send a task
    for i := 0; i < 2; i++ {
        go worker(jobs)
    }

    for i := 0; i < 10; i++ {
        job := make(chan int)
        select {
            case jobs <- job:
                fmt.Println("reserve worker for task:", i)
                job <- i // build and send the task
            default:
                fmt.Println("drop task:", i)
        }
    }
}

func worker(jobs chan chan int) {
    for job := range jobs {
        task := <- job
        fmt.Println("done task:", task)
    }
}
```

# 12/17 Prefer non-blocking sending

```
select {
case ch1 <- "value1": // strategy 1
case ch2 <- "value2": // strategy 2
default:
    // take an action
}
```

- No `time.After` to control the pool
- Use buffered channels for backpressure

## 13/17 Requirements for the pool

- Worker must know how to restart itself
- Fixed size of the pool and adjust per instance type
- Send metrics to centralized location and keep the current state on the instance

What to profile:

- How long does it take to process the task
- How many workers are busy right now
- What was the maximum number of concurrently occupied workers during lifetime of the process

# 14/17 Single buffer

```
type A struct { B int; _ int }

func main() {
    profile(func(){
        for i := 0; i < 10; i++ {
            list := make([]*A, 1000)
            for i := 0; i < 1000; i++ {
                list[i] = &A{B: i}
            }
        }
    })
}

func profile(fn func()){
    o := new(runtime.MemStats)
    runtime.ReadMemStats(o)
    fn()
    n := new(runtime.MemStats)
    runtime.ReadMemStats(n)
    fmt.Printf("objects %v alloc %v", n.HeapObjects-o.HeapObjects, n.HeapAlloc - o.HeapAlloc)
}
```

Run

# 15/17 Double buffers

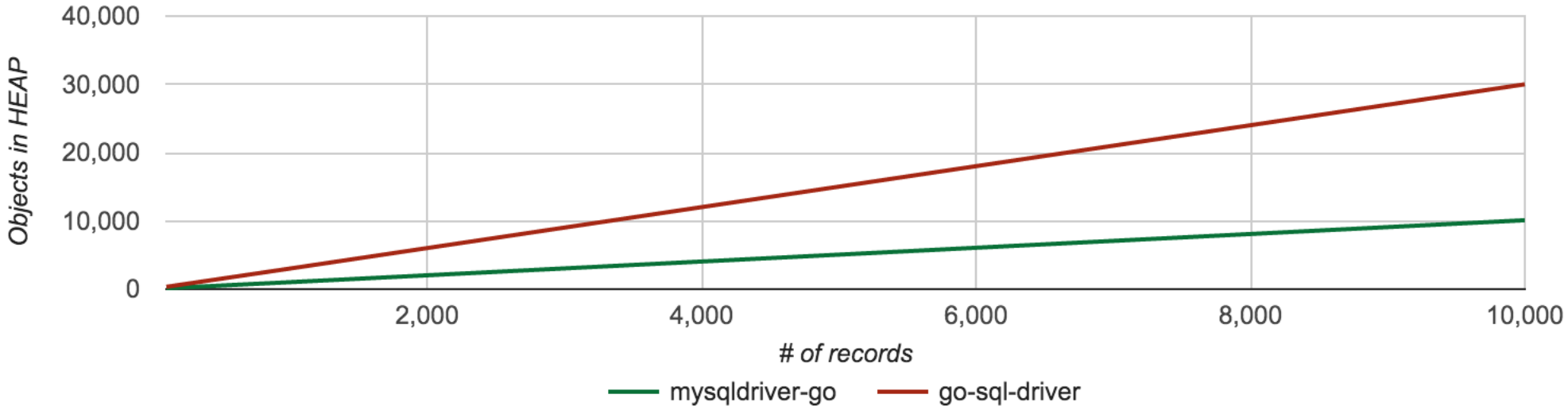
```
func main() {
    profile(func(){
        lists := make([][]*A, 2)
        lists[0] = make([]*A, 1000)
        lists[1] = make([]*A, 1000)

        for x := 0; x < 5; x++ {
            for y := 0; y < 1000; y++ {
                a := lists[0][y]
                if a == nil {
                    a = &A{}
                }
                a.B = x*y
                lists[0][y] = a
            }

            lists[0], lists[1] = lists[1], lists[0]
        }
    })
}
```

Run

# 16/17 Sometimes we have to write our library

pubnative/mysqldriver-go (https://github.com/pubnative/mysqldriver-go) GC optimized MySQL driver

pubnative/mysqlproto-go (https://github.com/pubnative/mysqlproto-go) Heap friendly implementation of the MySQL protocol



# 17/17 Keep the test code in one function

```
func TestMyFunc(t *testing.T) {
    // 1. setup test
    // 2. define and execute test cases
    // 3. cleanup
}
```

- Keep tests straightforward, reading from top to bottom you should be able to understand how the code performs and what are requirements for it
- No helpers and abstractions in the test code
- Define what's required to execute the code you test, nothing more

# Thank you

Kostiantyn Stepaniuk
PubNative GmbH
kostiantyn@pubnative.net (mailto:kostiantyn@pubnative.net)