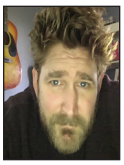
# iVoyeur
## Go Instrument Some Stuff

DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

I hate talking about programming languages. Have you heard of Alcibiades? He was, well, I guess you could say he was the frat boy of ancient Greece. The original Bro. There's this pretty funny story about him where he was wrestling this other guy (I forget his name), and Alcibiades, feeling that he was about to lose, bit him on the ear. Yes, exactly like Mike Tyson.

Now biting your wrestling partner was, in ancient Greece, every bit as frowned upon as it is today. It is not, suffice to say, a statutorily valid attack in wrestling among gentlepersons. And Alcibiades' opponent didn't have any qualms about letting him know; his quote (according to Plutarch, who wasn't there) was:

"Alcibiades! You bite! Like a woman!"

Setting aside for the moment the Grecian misogyny—for which I apologize on their behalf (as a male, not as a Grecian)—I think Alcibiades' opponent was expressing a few overlapping emotions here. There was, of course, the outrage at having been bitten (especially in the course of, no doubt, well-executed by-the-book wrestling on his part). And then there was the surprise at having been bitten by Alcibiades, who was (despite ample evidence to the contrary) always assumed to be a stand-up bro by those who hung out with him. And, finally, there was the shaming component of the accusation, the part where he called Alcibiades out by comparing him to the so-called "weaker sex."

I'm sure everyone in attendance thought this was a pretty slick burn. I can almost hear the room explode with the ancient Grecian analog of "Oh *snap*!" And I mean he deserved it, right? Surely everyone could agree that biting was not only illegal, but also un-bro-like, which put Alcibiades clearly in violation of not only the wrestling rule book, but also, and probably more importantly, bro-code—or whatever you want to call that unwritten collection of etiquette particular to those people in that place. The former violation merely lost him the match, but it was the latter that made him worthy of disdain. But *then* Alcibiades replies:

"Nay. I bite like a *lion*!"

He did this sort of thing all the time; just running roughshod over the rules and undermining anyone who called him on it. It was basically his thing, and he did it so confoundingly well that he *always* got away with it. He just didn't consider defying convention something to be ashamed of and was therefore immune to this sort of politesse-rooted shaming.

### Programming Languages

When I talk about programming languages, I always wind up feeling just exactly like the guy Alcibiades bit must have felt, which is to say: pretty sure what I just said was technically correct, but no longer convinced that it matters, and therefore confused about my place in the universe.

I'm not possessed of a Herculean-strength intellect, and I struggle to learn these languages just like the guy Alcibiades bit no doubt worked hard to master wrestling. Everyone assured us both it was the right and proper thing to do (it says so right there in the introduction sec-

tion of the O'Reilly book you wrote about the awesome new language you designed). And like the guy Alcibiades bit, I maintain this assumption that we, the community of people who struggled to learn Ruby or Python or Java or wrestling or whatever, have an understanding about what it means to be "good" or "bad" when we go about it. About the merits of this or that programming philosophy. About what constitutes an acceptable degree of inefficiency. About what is and is not secure.

But really, we don't have anything like that understanding. And bite by bite, I'm slowly beginning to realize that I will never have whatever equates to moral high-ground with respect to programming languages. That in fact maybe there never was such a thing, it just looked that way because my world was so small. There will never be *that* language that everyone who uses programming languages can finally maintain at least a begrudging respect for. Maybe that's a good thing, but it also means I am forever doomed to happily enter into excited conversation about this or that thing we're building only to be bitten on the ear over the language we chose to build it in.

I tire of this—this stupefied grasping at my bloody ear—and it's making me gun-shy. I've wanted to write this article for months, but I keep on balking because I know we're going to have to talk about languages. Well, at least one language, and worse, I'm going to have to *pick* it. I'm going to have to, once again, admit to liking a programming language, or at least admit to using one. My ear hurts just thinking about it.

## Instrumenting Golang

Oh well. Let's get this show on the road. I want to talk about instrumenting the programs you create. And I'm going to do it in Golang, so deal with it.

When I say instrument, I'm talking about actually placing code inside the things you write that is designed to either time an interaction or quantify how often something occurs. It's easier if I just show you.

To that end I've written a Web service. It's pretty typical of the sort of thing I do when I wear my Ops hat these days: a simple program that listens for HTTP GET requests on port 80 and exposes some bit of operational knowledge to whatever happens to be asking. This one responds with an "answer." Here's what an answer looks like:

```
type Answer struct {
  Type  string
  Desc  string
  Get   func(index int) string
  Rand  func() string
  DB    []string
}
```

Even if you don't speak Go, this should be pretty obvious: an answer is a data structure that consists of a type; a description; two functions, one for getting specific answers and another for getting random answers; and an array of strings where we keep all of our responses.

I won't have space to paste all of the code for this project here, but you can clone it from GitHub (https://github.com/djosephsen/answers), which means you can also `go get` it with `go get github.com/djosephsen/answers`. You'll find the source under the src directory in your `GOPATH`.

Since one likes to be modular about these things, the code is designed so that we can come along later and add new types of "answer" and register them into a global index of answers we can give. If you look in the root directory, you'll see that it comes with two types of answer modules, one that provides answers to the question "Why did the chicken cross the road?" and one that provides answers to the question "Knock-knock. Who's there?"

In `main.go`, you'll see that after we go about registering the available modules into the global index of answers with `initAnswers()`, we use the `net/http` module to register two Handler functions with `net/http` and then start listening on port 8080.

```
func main() {
  initAnswers()
  metrics.Connect()
  http.HandleFunc("/", helpHandler)
  http.HandleFunc("/get/", getHandler)
  http.ListenAndServe(":8080", nil)
}
```

Now, ignoring `metrics.Connect()` for a moment, I think you can kind of see where this is going. If a user gets '/', we respond with a help menu, but if you ask for something that begins with '/get/', we launch `getHandler()`.

The help handler traverses all the answers we know about and prints back a list of URLs of the form `/get/answer.Type` followed by the `answer.Desc` that corresponds to the Type. We can see what it looks like when we use curl to ask for '/'.

```
(osapi) [dave@otokami][librato-vagrant] [master|+ 1]
-> curl localhost:8080
Welcome to the answer service:
Valid answers:
/get/chicken :: Answers to why did the chicken cross the road?
/get/knocknock :: Knock Knock jokes1
```

Then when we ask for "/get/chicken" the getHandler function fires, and we get a random answer from the chicken module via that module's `Rand()` function. We can see what it looks like from `curl`:

```
(osapi) [dave@otokami][librato-vagrant] [master!.2+ 1]
-> curl localhost:8080/get/chicken
because the road crossed him
```

If you write Go, this code should be pretty familiar. It's a classic pattern for using `net/http` to write Web services in Go. In fact it began life as a copy/paste straight out of the `net/http` documentation. I'm not going to delve into the answer modules, because to instrument this application we don't need to leave `main.go`.

What we want to do is time and quantify our calls to the various handlers this program has now, as well as any that we might add in the future. In other words, we want to know how often `get/chicken` is called, and we want to know how long `get/chicken` takes to do what it does. And we don't just want that for `get/chicken`, we also want it for the help handler, `get/knocknock`, and any other answer modules we might add in the future. And we're lazy so we don't want to add new instrumentation every time we add another answer module.

And there's one more problem that you'll already be aware of if you're in the habit of timing function calls. Sometimes, aberrant measurements occur, such as calls to `get/chicken` that take three seconds because of bogons (possibly in the monitoring code) that we'll never be able to effectively track down. So we're going to want to generate percentiles for our timing measurements. We want to put extremely aberrant measurements in perspective. Is it one time in a million, or is it one time in a hundred?

What I'm very intentionally describing here is the use-case for which StatsD [1] was invented. We use it all over the place at Librato, and if you write services like these you probably should too unless you have something better. Specifically, we run Statsite [2] (a StatsD clone written in C (because native StatsD is a NodeJS daemon (lol programming languages))) on every instance we bring up. That way we can emit metrics directly from each instance to our metrics backend rather than risking packet-loss over the wire to a centralized StatsD instance (StatsD is a UDP protocol).

In this project, I'm using Etsy's `statsd` client, which is imported by my code in `metrics/metrics.go`. It provides a decent set of primitives but doesn't really give us what I'd call a Go-idiomatic means (lol programming languages) of implementing what we want here, so I have a few functions in `metrics.go` to help the client out. Let's take a look at my `Time()`:

```
func Time(name string, start time.Time) {
  if client == nil {
    return
  }
  now := time.Now()
  duration := now.Sub(start)
  if duration > 5000*time.Millisecond {
    fmt.Printf("Latent measurement for %q: %s", name, duration)
  }
  milliseconds := int64(duration / time.Millisecond)
  toStatsd(func() {
    // record the duration
    client.Timing(name, milliseconds)
    // also record a count
    client.UpdateStats([]string{fmt.Sprintf("%s.count", name)},
                                    1, 1)
  })
}
```

You'll find this function in several of our Go projects at Librato, and it's pretty clever. It takes a start-time and the name of the metric we want to show up in the metrics backend. It then computes the difference between the given start time and now and sends that duration into `statsd`. But wait, you're wondering, how does that time the actual function invocation? Stand by, that's the clever part. But before I get to that, you'll notice that it sends the timing to StatsD by way of a local `toStatsd()` function, which in turn takes a function with no arguments as its argument. Let's look at `toStatsd()`:

```
func toStatsd(fn func()) {
  start := time.Now()
  fn()
  duration := time.Now().Sub(start)
  if duration > 250*time.Millisecond {
    fmt.Printf("Statsd time took %s", duration)
  }
}
```

This was probably more along the lines of what you expected to see in `Time()`. This function captures the before time, runs the given function, and then captures the after time. If you look back up at `Time()`, you'll notice that we're passing `toStatsd()` an anonymous function that sends in the actual timing metric and increments a counter. So really `toStatsd()` is just a wrapper to time how long it takes the `statsd` client itself to do its thing. We're actually measuring how much latency `statsd` itself incurs.

Now let's bring this full circle by taking a look at `getHandler()` in `main.go` to see how we use `metrics.Time()`:

```
func getHandler(w http.ResponseWriter, r *http.Request) {
  answerType := r.URL.Path[len("/get/"):]
  defer metrics.Time("answer.handler."+answerType, time.Now())
  fmt.Fprintf(w, "%s\n",a.Answers[answerType].Rand())
}
```

Ah hah, the plot thickens. We're calling `metrics.Time` in a `defer` statement. If you don't program in Go, the `defer` statement is used to postpone the execution of a given function until just before the parent function returns. The interesting part about

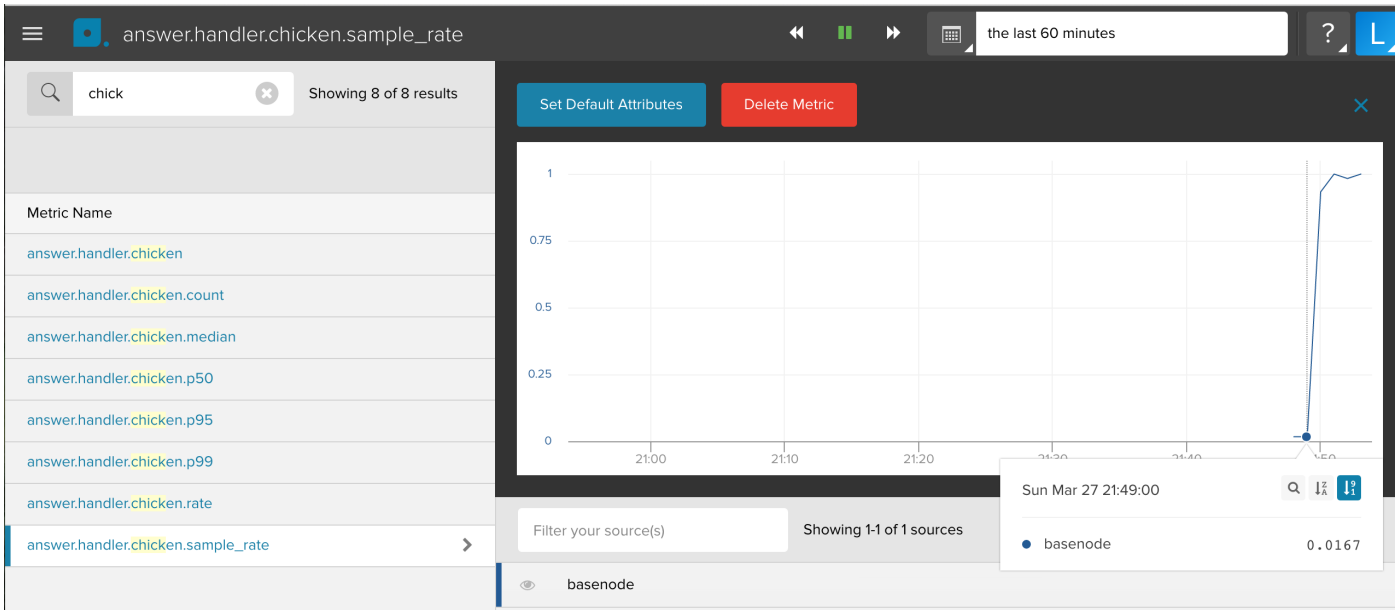



**Figure 1:** Look at all the lovely data!

this is that `defer` evaluates our functions *arguments* immediately, so when we pass `time.Now()` as the second argument to `metrics.Time`, that's evaluated immediately. That's how we capture our "before" time. Then `defer` takes care of executing `metrics.Time()` just before the function returns (after our answer module has done its thing), and as we've already seen `metrics.Time` captures its own after-time.

This gives us a single line of code we can inject at the beginning of any function in Go to get timing data as well as a count and rate of that function's invocation. The percentiles come automatically from StatsD as you can see in Figure 1.

So aside from `metrics/metrics.go`, which is completely reusable and modular/importable if desired, I've only added three lines of code to fully instrument every handler invocation this application has and any that might be added in the future, and one of those three lines was `metrics.Connect()`, which opens a socket to the local StatsD daemon.

I don't care how much you hate Golang, that's pretty cool, right? And, I mean, look at the graphs, *everybody* likes graphs … right?

Ow. My ear!

### *References*

[1] StatsD: https://github.com/etsy/statsd.

[2] Statsite: https://github.com/armon/statsite.