# Uptime: Building Resilient Services with GO

GopherCon 2015: Blake Caldwell - Uptime: Building Resilient Services with Go - YouTube

Was at Fog Creek Software on Kiln (Git & Mercurial source code hosting)

## Resiliency Tips

### Handle All Errors

#### `Nil` Checks

```
// can return nil in the resourceA
resourceA, err := OpenResourceA()
if err != nil {
  return nil, err
}
defer resourceA.Close() // will never panicl
```

Now we need to check for `nil` when `resourceA.Close()` gets called.

```
// Don't forget: resource might be nil!
func (resource *Resource) Close() {
  if resource != nil {
      // clean up
  }
}
```

### Channel Axioms

1. A send to a `nil` channel **blocks forever**
2. A receive from a `nil` channel **blocks forever**
3. A send to a closed channel **panics**
4. A receive from a closed channel returns the zero value immediately

http://dave.cheney.net/2014/03/19/channel-axioms

### Panics!

You can recover from panics

- … but this shouldn't be the standard
- Only recover if you're sure it's ok.
- Panic recovery is for current goroutine
- At very least, log the stack trace

### Avoid Race Conditions

- Reports when variable access is not synchronized
- Crashes with a full stack trace including the read and write goroutines
- Should be used in unit tests, development, and testing environments

```
go test -race mypkg    // to test the package
go run -race mysrc.go  // to run the source file
go build -race mycmd   // to build the command
go install -race mypkg // to install the package
```

### Implement Timeouts

Network Timeouts

- network dial timeout
- network connection inactivity timeout
- total connection timeout

### Test All The Things

- Integration tests work well with Docker

### Know Your Service

#### How Does It Use Memory? Profile It

What to watch

- How much memory does the service use when idle?
- How much memory per connection?
- Does the system reclaim memory that is no longer needed?
- What's the garbage collector doing? `GODEBUG=gctrace=1`
- Where is memory allocated? (PPROF)

#### PPROF

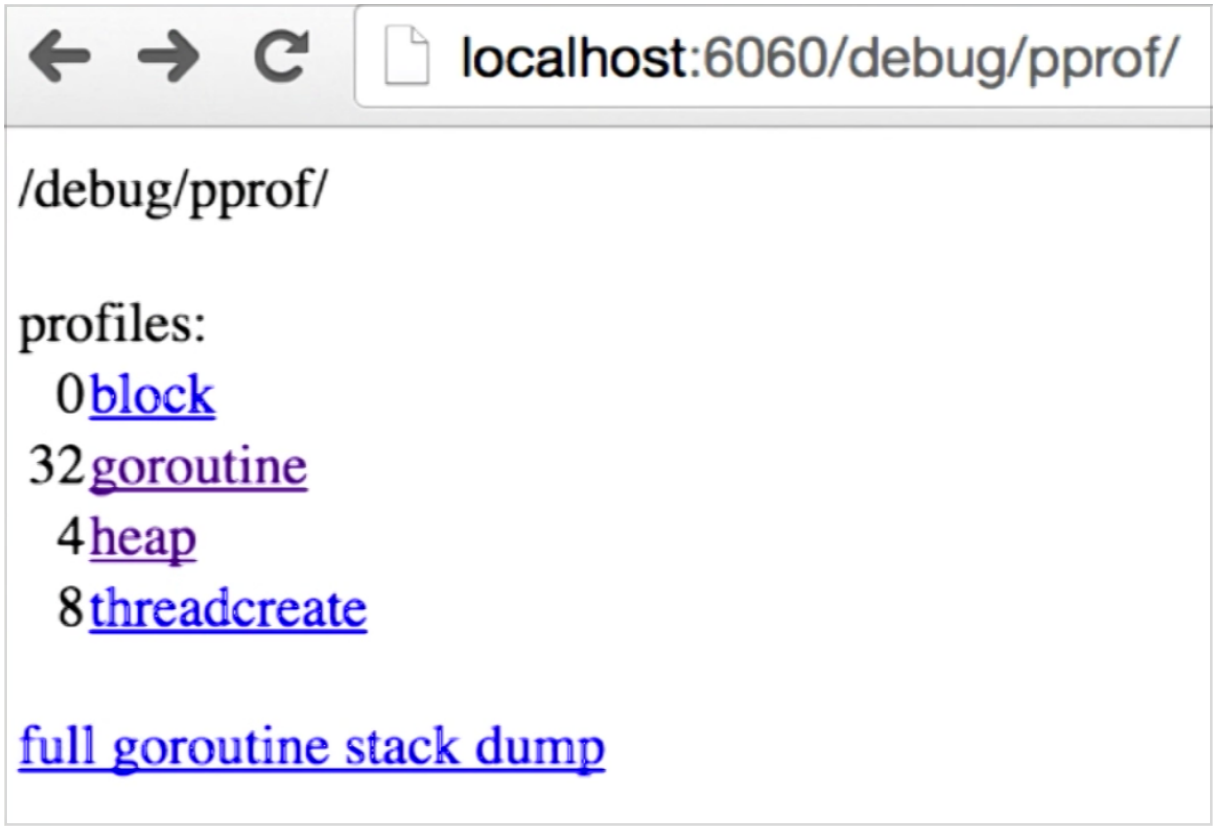
- Blocking Profile
- Goroutine count and full stacktraces
- Heap profile
- Stacktraces that lead to thread creations

Use it by importing the package `net/http/pprof`

```
import (
  _ "net/http/pprof"
  "net/http"
)

func main() {
  http.ListenAndServe(":6060", nil)
}
```

Then just load it up in the browser:




We can use PPROF to verify we are not leaking goroutines. It can tell you:

- How many goroutines when nobody is connected?
- How many goroutines per connection?
- Are all goroutines cleaned up after all connections close?

Clicking in gives you a stack trace of your LIVE code / service

```
go tool proof ./server http://localhost:6060/debug/pprof/goroutine

(pprof) top5

...

(pprof) web // pops up a browser showing you a SVG graphic
```

```
go tool proof ./server http://localhost:6060/debug/pprof/heap

(pprof) top5

... shows you the memory allocation

(pprof) web // pops up a browser showing you a SVG graphic
```

### Watch It Run

`/info` endpoint (internal only)

```
{
  Version: "1.0.275-b244a2b9b8-20150202.163449",
  StartTimeEpochSecs: 1430515329,
  CurrentTimeEpochSecs: 143117131,
  Uptime: "167h10m2s"
}
```

### Managing Service Version

Version Number:
`<major>.<minor>.<commit#>-<Git SHA>-<date>.<time>`

You store the service version in a global variable in your code:
`var ServiceVersion string`

Build script:

```
go build -ldfags "-X main.ServiceVersion 1.0.275-b244a2b9b8-20150202.163449" serviceCmd
```

### Keep Good Logs

- Created a semi-unique string per request
- Use this request string as a prefix in all log entries
- Always log at least the start and end of a request (see next section *why* this is good)

### Who is Currently Connected?

`/connections` endpoint

```
{
  "CurrentUserCount":1,
  "CurrentlyAuthenticatedUsers":
  [
      {
          "Account":"aviato",
          "Name":"...",
          "PublicKeyName":"BuildServer",
          "SessionKey":"106abc0c",
          "SessionDuration":"25m4s"
      }
  ]
}
```

### Drain and Die

Planned restart means serving all requests and stop listening for new requests. `SIGTERM` means gracefully allowing this to happen.

#go talks/2015/gophercon# #people/blake caldwell#