An Elaborate Web Server

This lesson covers all the primary services provided by a server by implementing a detailed web server, and showing how each service is delivered independently.

WE'LL COVER THE FOLLOWING

To further deepen your understanding of the `Http` package and how to build web server functionality, study and experiment with the following example. First, the code is listed, then, the usage of the different functionalities of the program and its output are shown.

The example below demonstrates various ways in which you can write web server handlers. The web server itself is started at **line 43** in `main()` with error-handling in the following lines.

Package `expvar` provides a standardized interface to public variables, such as operation counters in servers. A variable containing the number of hello requests is defined at **line 15**, and a struct `Counter` at **line 22** as well as a channel of integers at **line 27**. Also, some command-line flags are defined (from **line 17** to **line 19**, which are then parsed at **line 30**.

Then, from **line 31** to **line 42**, we define all of the handlers, which are used by this webserver. At **line 34**, we define a new `Counter` variable named `ctr`. Then, at **line 36**, we publish `ctr`, which is global to the webserver.

More detailed explanations of each of the web server handlers can be found below, which also shows you which request URL corresponds to which handler function.

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
        "bytes"
        "expvar"
        "flag"
        "fmt"
        "net/http"
        "io"
        "log"
        "os"
        "strconv"
)

// hello world, the web server
var helloRequests = expvar.NewInt("hello-requests")
// flags:
var webroot = flag.String("root", "/home/user", "web root directory")
// simple flag server
var booleanflag = flag.Bool("boolean", true, "another flag for testing")

// Simple counter server. POSTing to it will set the value.
type Counter struct {
        n int
}

// a channel
type Chan chan int

func main() {
        flag.Parse()
        http.Handle("/", http.HandlerFunc(Logger))
        http.Handle("/go/hello", http.HandlerFunc(HelloServer))
        // The counter is published as a variable directly.
        ctr := new(Counter)
        expvar.Publish("counter", ctr)
        http.Handle("/counter", ctr)
```

```
        // http.Handle("/go/", http.FileServer(http.Dir("/tmp"))) // uses the OS filesystem
        http.Handle("/go/", http.StripPrefix("/go/", http.FileServer(http.Dir(*webroot))))
        http.Handle("/flags", http.HandlerFunc(FlagServer))
        http.Handle("/args", http.HandlerFunc(ArgServer))
        http.Handle("/chan", ChanCreate())
        http.Handle("/date", http.HandlerFunc(DateServer))
        err := http.ListenAndServe("0.0.0.0:3000", nil)
        if err != nil {
                log.Panicln("ListenAndServe:", err)
        }
}

func Logger(w http.ResponseWriter, req *http.Request) {
        log.Print(req.URL.String())
        w.WriteHeader(404)
        w.Write([]byte("oops"))
}

func HelloServer(w http.ResponseWriter, req *http.Request) {
        helloRequests.Add(1)
        io.WriteString(w, "hello, world!\n")
}

// This makes Counter satisfy the expvar.Var interface, so we can export
// it directly.
func (ctr *Counter) String() string { return fmt.Sprintf("%d", ctr.n) }

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
        switch req.Method {
        case "GET": // increment n
                ctr.n++
        case "POST": // set n to posted value
                buf := new(bytes.Buffer)
                io.Copy(buf, req.Body)
                body := buf.String()
                if n, err := strconv.Atoi(body); err != nil {
                        fmt.Fprintf(w, "bad POST: %v\nbody: [%v]\n", err, body)
                } else {
                        ctr.n = n
                        fmt.Fprint(w, "counter reset\n")
                }
        }
        fmt.Fprintf(w, "counter = %d\n", ctr.n)
}

func FlagServer(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("Content-Type", "text/plain; charset=utf-8")
        fmt.Fprint(w, "Flags:\n")
        flag.VisitAll(func(f *flag.Flag) {
                if f.Value.String() != f.DefValue {
                        fmt.Fprintf(w, "%s = %s [default = %s]\n", f.Name, f.Value.String(),
                } else {
                        fmt.Fprintf(w, "%s = %s\n", f.Name, f.Value.String())
                }
        })
}

// simple argument server
func ArgServer(w http.ResponseWriter, req *http.Request) {
        for _, s := range os.Args {
                fmt.Fprint(w, s, " ")
        }
```

```
}

func ChanCreate() Chan {
        c := make(Chan)
        go func(c Chan) {
                for x := 0; ; x++ {
                        c <- x
                }
        }(c)
        return c
}

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
        io.WriteString(w, fmt.Sprintf("channel send #%d\n", <-ch))
}

// exec a program, redirecting output
func DateServer(rw http.ResponseWriter, req *http.Request) {
        rw.Header().Set("Content-Type", "text/plain; charset=utf-8")
        r, w, err := os.Pipe()
        if err != nil {
                fmt.Fprintf(rw, "pipe: %s\n", err)
                return
        }

        p, err := os.StartProcess("/bin/date", []string{"date"}, &os.ProcAttr{Files: []*os.Fi
        defer r.Close()
        w.Close()
        if err != nil {
                fmt.Fprintf(rw, "fork/exec: %s\n", err)
                return
        }
        defer p.Release()
        io.Copy(rw, r)
        wait, err := p.Wait()
        if err != nil {
                fmt.Fprintf(rw, "wait: %s\n", err)
                return
        }
        if !wait.Exited() {
                fmt.Fprintf(rw, "date: %v\n", wait)
                return
        }
}
```

> **Remark:** Change **line 43** to `err := http.ListenAndServe(":12345", nil)`, if you're running it locally.

Response in browser

Copy your URL written at the bottom of code widget next to **Your app can be found at**. For example, https://1dkne4jl5mmmm.educative.run

Logger

https://1dkne4jl5mmmm.educative.run/ (root)

oops

On *Windows*: A dialog window pops up saying: `Unable to open .../. The internet site reports that the item you requested could not be found (HTTP 1.0 / 404)`.

The Logger prints a **404 Not Found** header with `w.WriteHeader(404)`. This technique is generally useful whenever an error is encountered in the web processing code; it can be applied like this:

```
if err != nil {
  w.WriteHeader(400)
  return
}
```

It also prints **date** + **time** through the logger function and the URL for each request on the command window of the web server.

> **Remark:** To test it locally, try http://localhost:12345/.

HelloServer

https://1dkne4jl5mmmm.educative.run/go/hello

hello, world!

The package `expvar` makes it possible to create variables (of types Int, Float, String) and make them public by publishing them. It exposes these variables via HTTP at /debug/vars in JSON format. It is typically used for, e.g., operation counters in servers. The `helloRequests` is such an int64 variable; this handler adds 1 to it and then writes **hello, world!** to the browser.

> **Remark:** To test it locally, try http://localhost:12345/go/hello.

Counter

https://1dkne4jl5mmmm.educative.run/counter

counter = 1

refresh (= GET)

counter = 2

Each refresh increments the counter. The counter object `ctr` has a String() method and implements the `Var` interface. This makes it publishable even though, it is a struct. The function `ServeHTTP` is a handler for `ctr` because it has the right signature.

> **Remark:** To test it locally, try http://localhost:12345/counter.

FileServer

https://1dkne4jl5mmmm.educative.run/go/ggg.html

404 error

On *Windows*: A dialog window pops up saying: `Unable to open the page .../ggg.html. The internet site reports that the item you requested could not be found (HTTP 1.0 / 404)`

FileServer returns a handler that serves HTTP requests with the contents of the file system rooted at root. To use the operating system's file system, use `http.Dir`, as in:

```
http.Handle("/go/", http.FileServer(http.Dir("/tmp")))
```

> **Remark:** To test it locally, try http://localhost:12345/go/ggg.html.

FlagServer

https://1dkne4jl5mmmm.educative.run/flags

Flags:

boolean = true

root = /home/rsc

This handler uses the `flag.VisitAll` function to loop through all the flags, and prints their name, value and default value (if different from value).

> **Remark:** To test it locally try http://localhost:12345/flags.

ArgServer

https://1dkne4jl5mmmm.educative.run/args

./elaborated_webserver On Windows: *elaborated_webserver*

This handler loops through `os.Args` to print out all the command-line arguments. If there are none, only the program name (the path to the executable) is printed.

> **Remark:** To test it locally, try http://localhost:12345/args.

Channel

https://1dkne4jl5mmmm.educative.run/chan

channel send #1

refresh

channel send #2

Each refresh increments the channel #. The channel's `ServeHTTP` method shows the next integer from the channel at each new request. So a web server can take its response from a channel, populated by another function (or even a client). The following snippet shows a handler function which does exactly that but also times out after 30 s:

```
func ChanResponse(w http.ResponseWriter, req *http.Request) {
  timeout := make (chan bool)
  go func () {
    time.Sleep(30e9)
    timeout <- true
```

```
  }()
  select {

    case msg := <-messages:
    io.WriteString(w, msg)
    case stop := <-timeout:
    return
  }
}
```

> **Remark:** To test it locally try http://localhost:12345/chan.

DateServer

https://1dkne4jl5mmmm.educative.run/date

shows the current datetime (works only on Unix because it calls /bin/date)

Possible output: *Thu Sep 8 12:41:09 CEST 2011*

The `os.Pipe()` returns a connected pair of Files, reads from `r`, and returns bytes written to `w`. It returns the files and an error, if any, using:

```
func Pipe() (r *File, w *File, err error)
```

Timing your Http requests

The following function `makeLoggingHandler` enables you to do this:

```
func makeLoggingHandler(h http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {

    start := time.Now()
    h(w, r)

    log.Printf("%s\t%s\t%s", r.RemoteAddr, r.URL, time.Sub(start))

  }
}
```

Call it as: `makeLoggingHandler(yourHandler)`.

> **Remark:** To test it locally, try http://localhost:12345/date.

That's pretty much it about making an elaborate web server and its applications. In the next lesson, you'll learn how to send remote calls within different programs using Go.