Making a Web Application Robust

This lesson shows how to make an application robust by making it able to withstand a minor panic.

WE'LL COVER THE FOLLOWING

When a handler function in a web application panics our web server simply terminates. This is not good: a web server must be a robust application, able to withstand what perhaps is a temporary problem. A first idea could be to use `defer/recover` in every handler-function, but this would lead to much duplication of code. Applying the error-handling scheme with closures is a much more elegant solution. Here, we show this mechanism applied to the simple web-server made previously, but it can just as easily be applied in any web-server program.

To make the code more readable, we create a function type for a page handler-function:

```go
type HandleFnc func(http.ResponseWriter,*http.Request)
```

Our `errorHandler` function applied here becomes the function `logPanics`:

```go
func logPanics(function HandleFnc) HandleFnc {
  return func(writer http.ResponseWriter, request *http.Request) {
    defer func() {
      if x := recover(); x != nil {
        log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
      }
    }()
  function(writer, request)
 }
}
```

And we wrap our calls to the handler functions within `logPanics`:

```
http.HandleFunc("/test1", logPanics(SimpleServer))
http.HandleFunc("/test2", logPanics(FormServer))
```

The handler-functions should contain *panic* calls or a kind of `check(error)` function.

Example

The complete code is listed here:

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 (
"net/http"
"log"
"io"
)

type HandleFnc func(http.ResponseWriter,*http.Request)

const form = `<html><body><form action="#" method="post" name="bar">
<input type="text" name="in"/>
<input type="submit" value="Submit"/>
</form></html></body>`

/* handle a simple get request */
func SimpleServer(w http.ResponseWriter, request *http.Request) {
  io.WriteString(w, "<h1>hello, world</h1>")
}

/* handle a form, both the GET which displays the form
and the POST which processes it.*/
func FormServer(w http.ResponseWriter, request *http.Request) {
  w.Header().Set("Content-Type", "text/html")

  switch request.Method {
    case "GET":
      /* display the form to the user */
      io.WriteString(w, form );
    case "POST":
      /* handle the form data, note that ParseForm must
      be called before we can extract form data with Form */
      // request.ParseForm();
      //io.WriteString(w, request.Form["in"][0])
      // easier method:
      io.WriteString(w, request.FormValue("in"))
  }
}

func main() {
  http.HandleFunc("/test1", logPanics(SimpleServer))
  http.HandleFunc("/test2", logPanics(FormServer))
  if err := http.ListenAndServe("0.0.0.0:3000", nil); err != nil {
    panic(err)
  }
}

func logPanics(function HandleFnc) HandleFnc {
  return func(writer http.ResponseWriter, request *http.Request) {
  defer func() {
    if x := recover(); x != nil {
      log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
    }
  }()
  function(writer, request)
  }
}
```

Follow the same procedure to run the program discussed previously when making a web application.

> **Remark**: Change **line 42** to `if err := http.ListenAndServe(":8088", nil); err != nil {` if you're running it locally. And try URLs http://localhost:8088/test1 and http://localhost:8088/test2.

A better alternative to writing an application is using templates. In the next lesson, there is a detailed discussion on how to write an application using HTML templates.