

CH15 建立HTTP伺服器

15-1 前言

- 在前一章，學到了如何和一個伺服器溝通和取得資料，現在我們要深入探討這些伺服器是如何打造的，特別是伺服器要如何以不同方式回應客戶端的請求
- 由於網路伺服器也使用HTTP協定，因此可稱為HTTP伺服器；當我們用網頁瀏覽器瀏覽網站時，它會連結到一個HTTP伺服器和取回HTML網頁，然後顯示成我們能觀看的形式

- 也有些HTTP伺服器的目的是提供API讓其他程式呼叫；比如你在一些網站註冊時，會被詢問是否要用FB或google帳號登入，這表示該網站會呼叫FB或google API 來取得你這些帳號的資訊
 - 這些API通常傳回JSON或其他格式的結構化文字訊息，以便其他程式解讀，不是給人類閱讀的
- 有種網路的API是所謂的 RESTful API, 對於呼叫方式和HTTP請求有一定的風格要求，而且接收的資料型式是帶有參數的URL(即GET請求)，如今很常用來建置微服務(microservice)，本章就會來看如何打造最簡單的 RESTful API

15-2 打造最基本的伺服器

- 我們能寫出最基本的HTTP伺服器叫做 **Hello World 伺服器**：當使用者存取伺服器的位址時，它只會傳回一句文字 “**Hello World**”，沒有其他功能
- 這種伺服器沒有什麼用，不過能為後面更複雜伺服器的學習鋪路
- 下面就來看看這樣的伺服器要怎麼寫，並如何在普通網頁瀏覽器中(客戶端)顯示結果

15-2-1 使用HTTP請求處理器(handler)

- 為了應付HTTP請求，我們需要撰寫一個功能來處理請求，因此我們會把這個功能叫做handler (請求處理器)
- 在go語言有幾種方式能寫請求處理器，其中一種是實作 http 套件的 Handler 介面，這個介面只有一個方法 `ServeHTTP()`：

`ServeHTTP(w http.ResponseWriter, r *http.Request)`

- 這個方法會接收一個`http.Request`型別，也就是來自客戶端的請求，我們在前一章看到可以從他的標頭和主體讀取資料；而另一個參數則是`http.ResponseWriter`型別，即為要寫給客戶端的回應

- 為了實作HTTP請求處理器，可以先建立一個空結構，然後掛上ServeHTTP()方法：

```
type myHandler struct {}  
func(h myHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {}
```

- 這樣一來，上面的myHandler便是個合法的HTTP請求處理器，你就能使用http套件的ListenAndServe()函式來監聽(listen)指定的TCP位址：

```
http.ListenAndServe(":8080", myHandler{})
```

- 此舉等於啟動伺服器，監聽 http://localhost:8080和等待客戶端送出請求，若收到就會呼叫myHandler.ServeHTTP()

- 此外，`ListenAndServe()`函式有可能傳回一個**error**，我們在這種狀況下可能會希望報錯並終止程式，常見的方法是把这个函式包在 `log.Fatal()`內：

```
log.Fatal(ghttp.ListenAndServe(":8080, myHandler{}"))
```

練習：建立Hello World 伺服器

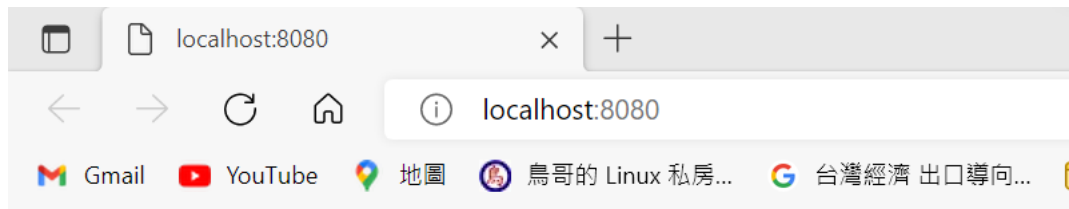
```
1  package main
2
3  import (
4      "log"
5      "net/http"
6  )
7
8  type hello struct{} //HTTP請求處理器
9
10 //請求處理器的方法實作
11 func (h hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
12     msg := "<h1>Hello World</h1>" //有HTML標籤的文字
13     w.Write([]byte(msg)) //寫入回應 (傳給客戶端)
14 }
15
16 func main() {
17     //啟動服務
18     log.Fatal(http.ListenAndServe(":8080", hello{}))
19 }
20
```


執行程式：

- 到主控台執行這支程式：

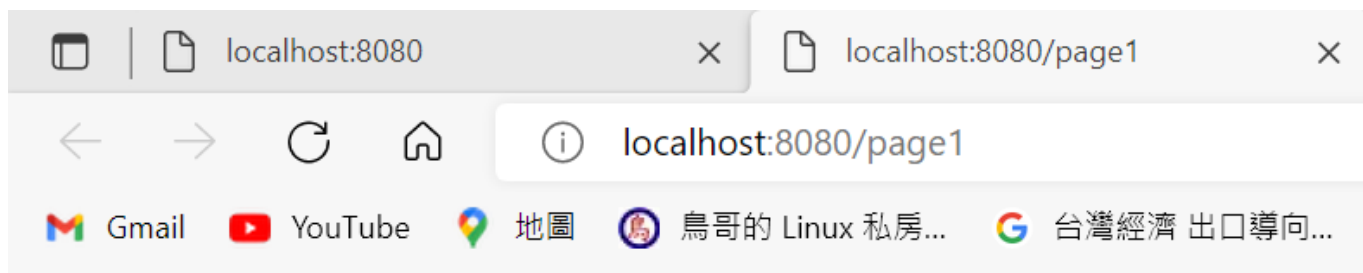
```
PS D:\git\Golang\ch15\15-2-1> go run .  
[]
```

- 接著打開瀏覽器，輸入網址 <http://localhost:8080>：



Hello World

- 有趣的是，若你改變路徑，比如輸入 <http://localhost:8080/page1>，c 還是會收到一樣的訊息：



Hello World

15-2-2 簡單的routing(路由)控制

- 在前面簡單的伺服器中，就算客戶端在伺服器路徑下加入不同的路徑，回應也永遠一樣
- 然而，不同的子路徑可能代表不同意義，比如：

<http://localhost:8080> ← 首頁

<http://localhost:8080/content> ← 目錄頁

<http://localhost:8080/page1> ← 第一頁

- 為了讓 參數pattern是要處理的子路徑，第二個參數handler則是此路徑被請求時要呼叫的函式，它必須有一個ResponseWriter和一個Request參數，和前面的ServeHTTP()方法相同

練習：讓伺服器處理路徑

```
1  package main
2
3  ∨ import (
4      |     "log"
5      |     "net/http"
6  )
7
8  type hello struct{}
9
10 //原本的請求處理器方法
11 ∨ func (h hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
12     |     msg := "<h1>Hello World</h1>"
13     |     w.Write([]byte(msg))
14 }
15
16 //新函式，用來處理對路徑 /page1的請求
17 ∨ func servePage1(w http.ResponseWriter, r *http.Request) {
18     |     msg := "<h1>Page 1</h1>"
19     |     w.Write([]byte(msg))
20 }
21
22 ∨ func main() {
23     |     //在客戶端請求路徑/page1 時呼叫servePage1
24     |     http.HandleFunc("/page1", servePage1)
25     |     //監聽localhost:8080 並在需要時呼叫hello.ServeHTTP()
26     |     log.Fatal(http.ListenAndServe(":8080", hello{}))
27 }
28
```

執行程式：

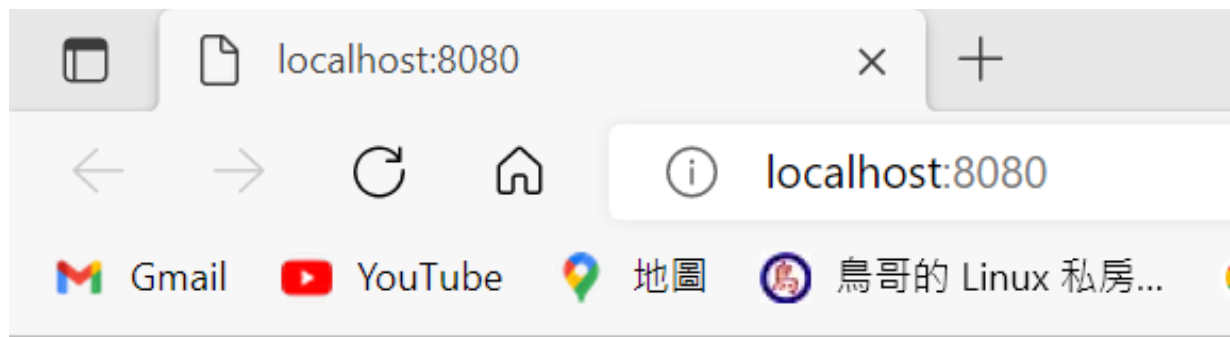
- 存檔後執行：

```
PS D:\git\Golang\ch15\15-2-2> go run .  
[]
```

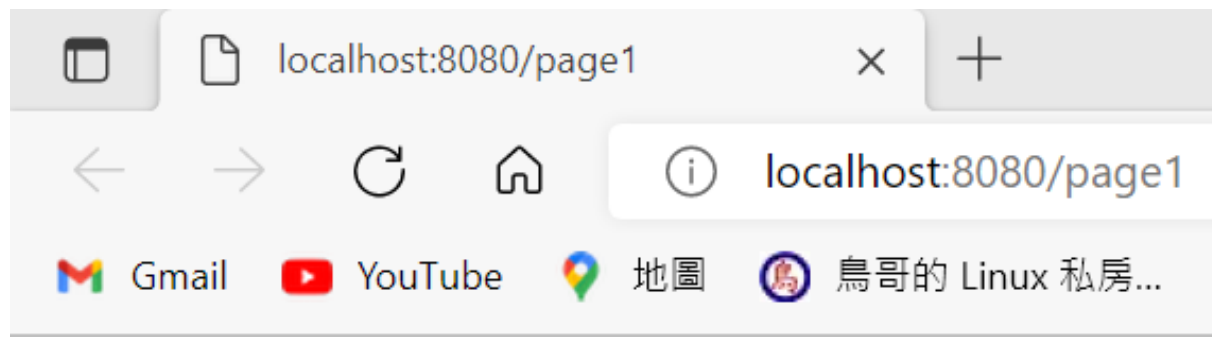
- 然後在瀏覽器輸入：

<http://localhost:8080>

<http://localhost:8080/page1>



Hello World



Hello World

- 怎麼還是沒有變化呢？因為對`http.HandleFunc()`來說，它設定的對象是http套件的`DefaultServeMux`結構
- `DefaultServeMux`是http套件預設的`ServeMux`結構，功能和我們自己定義的`hello`結構一樣
- 可是當我們要程式監聽請求時，卻指定用結構`hello`為請求處理器，那`DefaultServeMux`就不會發揮功能了

15-2-3 修改程式來應付多重路徑請求

- 以上問題的解決方式是統一使用DefaultServeMux來處理客戶端請求，並將hello結構註冊給DefaultServeMux作為請求處理器：

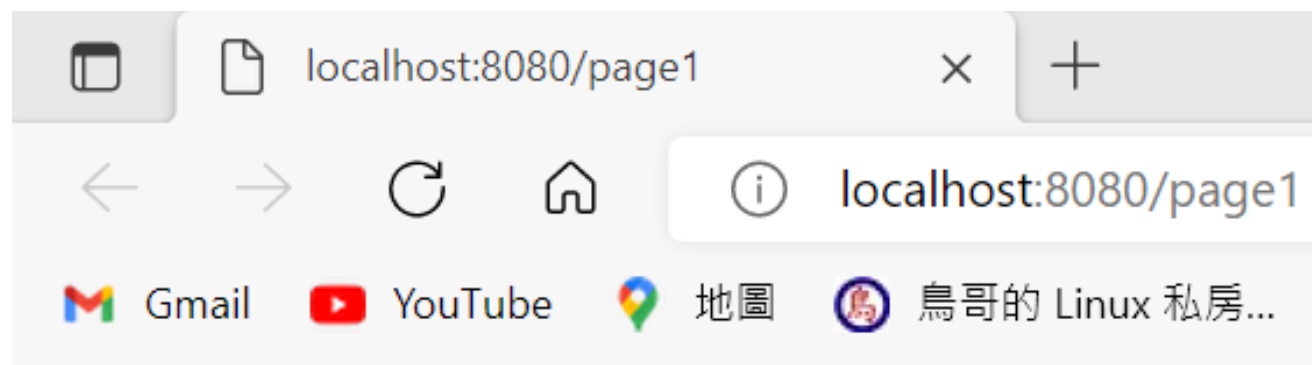
```
func Handle(pattern string, handler Handler)
```

- **pattern**參數代表請求處理器要負責的路徑，**handler**參數則是請求處理器結構

- 接著，當我們使用ListenAndServe()啟動伺服器時，它的第二個參數要設為nil，這樣才能用http.DefaultServeMux來監聽請求
- 所以上一個練習的main函式可以修改為如下：

```
func main() {  
    http.HandleFunc("/page1", servePage1)  
    http.Handle("/", hello{})  
    log.Fatal(http.ListenAndServe(":8080", nil))  
}
```

- 修改後重新執行伺服器，這回 `http://localhost:8080/page1` 順利回傳了新訊息



Page 1

使用自訂的ServeMux結構

- 你也可以使用自訂的ServeMux結構，它同樣擁有 `HandleFunc()` 和 `Handle()` 方法
- 以下是另一個改寫，建立一個稱為mux的結構，並使用它來接收客戶端請求：(執行效果和前面完全一樣, 只是使用mux而不是DefaultServeMux)

```
func main() {  
    mux := http.NewServeMux() //產生一個新的ServeMux  
    mux.HandleFunc("/page1", servePage1)  
    mux.Handle("/", hello{}) //把hello連同其ServeHTTP()方法註冊給mux  
    log.Fatal(http.ListenAndServe(":8080", mux)) //用mux來監聽請求  
}
```

回顧：請求處理器 vs. 請求處理函式

- 透過以上練習，應該可以注意，`http.Handle()`和`http.HandleFunc()`雖然都能處理特定路徑的請求，接收的參數卻不相同：

`http.Handle()`接收一個實作`http.Handler`介面的結構

`http.HandleFunc()` 接收一個函式

- 兩者到頭來都會呼叫一個擁有`http.ResponseWriter` 和 `*http.Request` 參數的函式來處理請求，感覺差異不大，不過在開發結構複雜的專案時，選擇正確的作法就很重要，好確保專案能採用做合適的架構
- 一般來說，若專案簡單，應該使用`http.HandleFunc()` 和套個簡單的函式，因為因此特地建一個結構就像是殺雞用牛刀了
- 但若你需要設定一些參數，或追蹤某些資料，把這些資料放在一個結構中就更適當

15-3 解讀網址參數來動態產生網頁

- HTTP伺服器可以根據更多的請求細節產生回應，而這些細節不僅能用路徑的形式，更可以用網址參數傳給伺服器
- 參數的傳遞有很多種，但最常見的還是使用查詢字串(QueryString)，它包含了所謂的查詢參數 (前一章開頭提過)

- 假設伺服器的URL為：<http://localhost:8080>
- 然後在後面加上一些字：<http://localhost:8080?name=john>
- ?name=john 就是所謂查詢字串，在此例中有一個參數name, 其值被指定為john
- 如果有更多參數，則會用&來連接：

<http://localhost:8080?name=john&age=30>

- 查詢字串通常搭配**GET**請求使用，因為**POST**請求一般會透過請求主體而非參數傳遞資料
- 若想解讀客戶端請求中URL夾帶的參數，要透過**http.Request** 結構URL 屬性的 **Query()** 方法
- 先來看看程式碼：

練習：顯示個人化的歡迎訊息

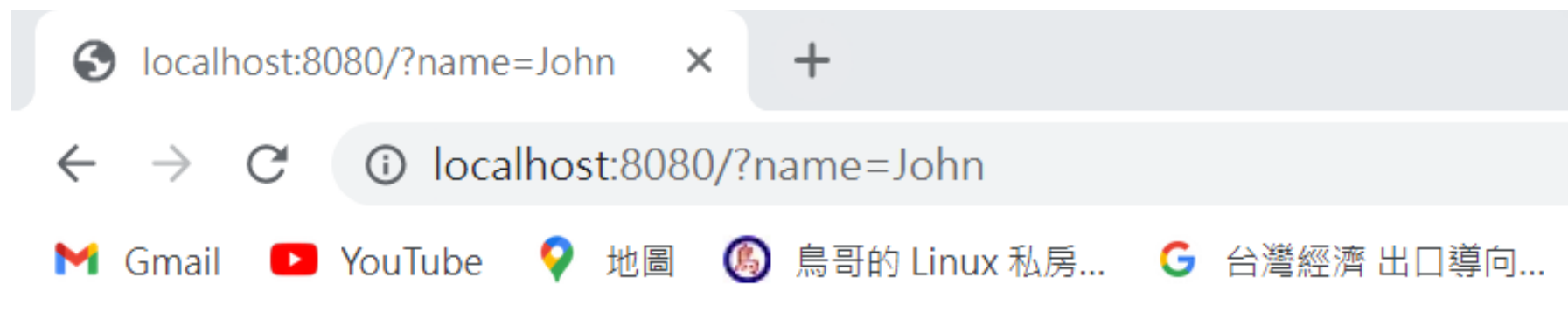
- 在這個練習中，我們要再來寫一個能顯示歡迎訊息的HTTP伺服器；但能夠讓使用者在URL後面加上name參數來傳入自己的名字，然後網頁頁面就會顯示 Hello XXX
- 若沒有提供名字，那伺服器會傳回HTTP狀態碼400 (bad request)

```
1  package main
2
3  import (
4      "fmt"
5      "log"
6      "net/http"
7      "strings"
8  )
9
10 func hello(w http.ResponseWriter, r *http.Request) {
11     vl := r.URL.Query()    //讀取查詢字串
12     name, ok := vl["name"] //讀取參數 name
13     if !ok {               //若查無參數
14         w.WriteHeader(http.StatusBadRequest) //回應HTTP 400
15         //在網頁產生針對使用者的歡迎訊息
16         w.Write([]byte("<h1>Missing name</h1>"))
17         return
18     }
19     w.Write([]byte(fmt.Sprintf("<h1>Hello %s</h1>", strings.Join(name, ","))))
20 }
21
22 func main() {
23     http.HandleFunc("/", hello)
24     log.Fatal(http.ListenAndServe(":8080", nil))
25 }
26
```

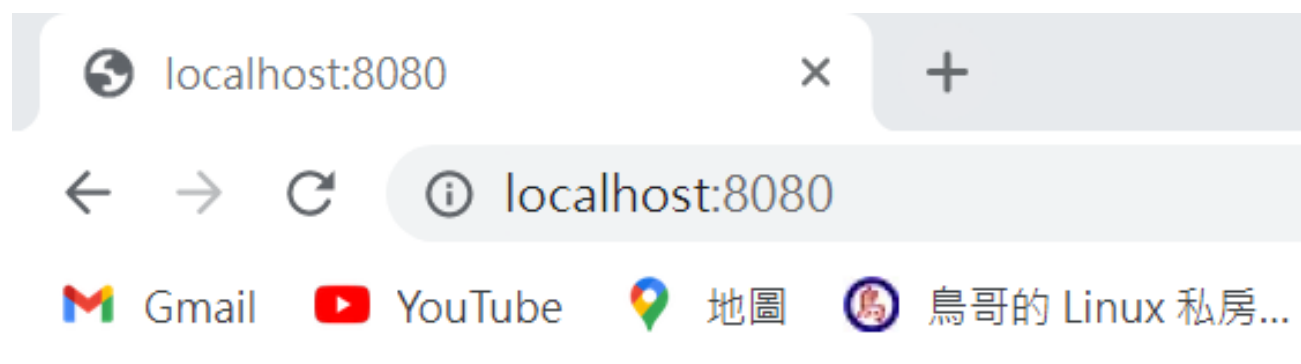
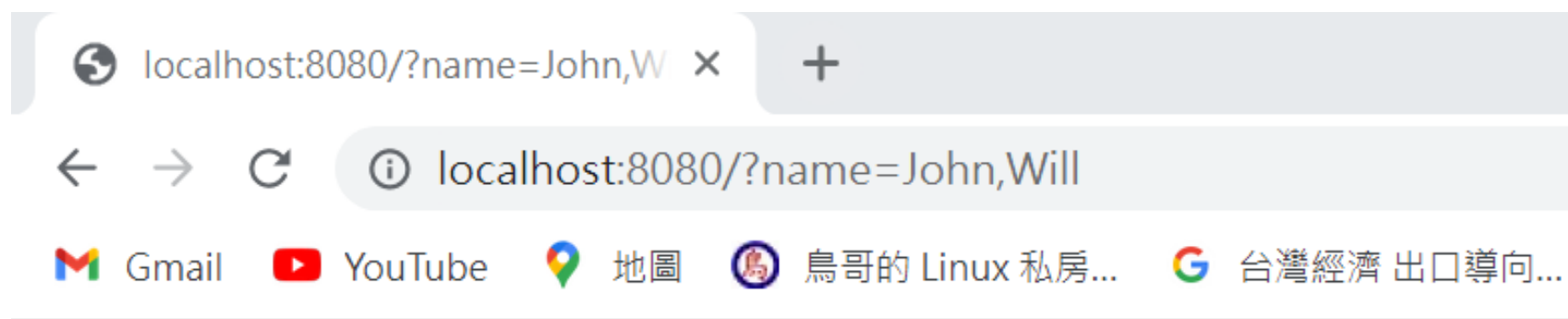
- `r.URL.Query()`會傳回`map[string][]string`型別，其鍵為參數名稱，對應值則為參數值
- 注意到參數值是個字串切片，因為使用者可能用 `?name=name1,name2` 的方式傳入不只一個名字
- 這便是為什麼要使用`string.Join()`將`name`切片內的元素連接起來，並以逗號連接成單一一一個字串，這麼一來，即使你輸入多重人名，程式也能解讀並正確顯示出來

執行結果：

```
PS D:\git\Golang\ch15\15-3> go run .  
[
```



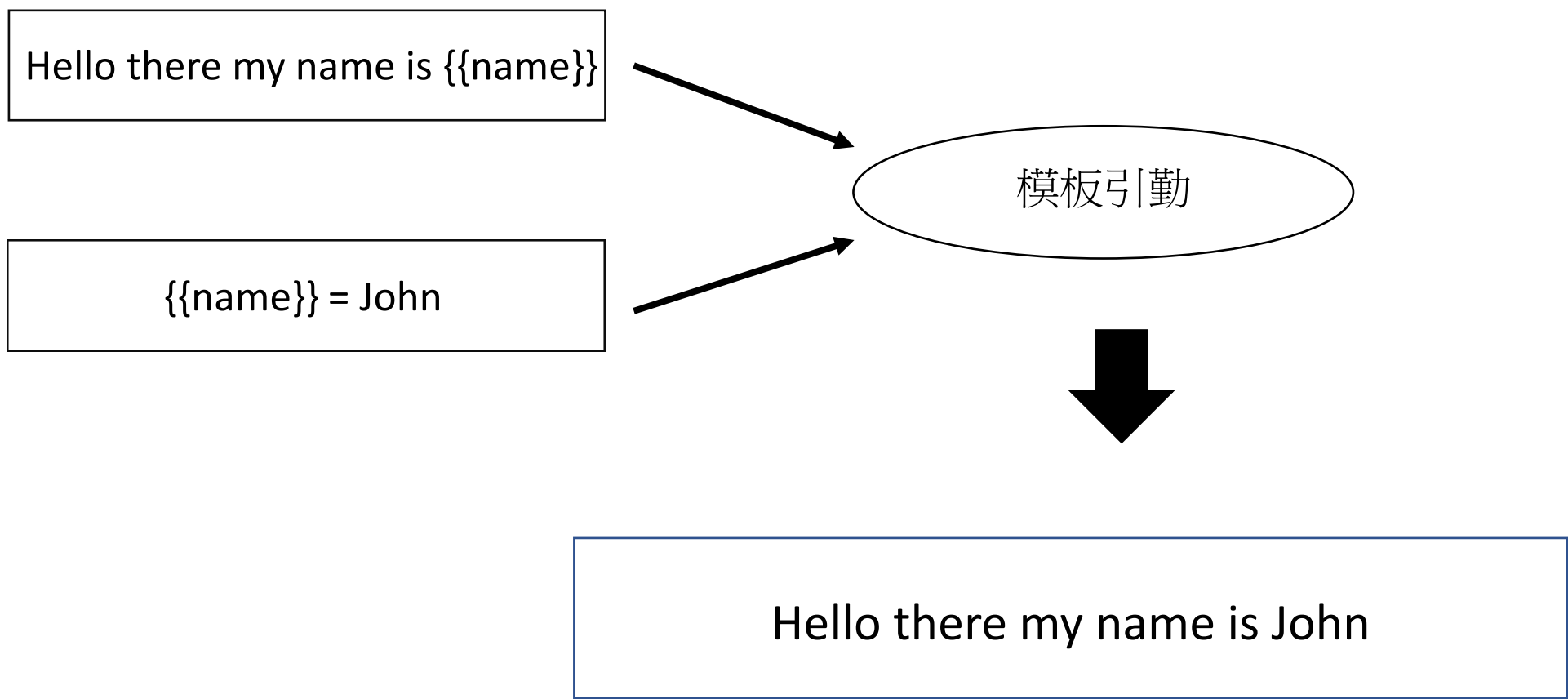
Hello John



15-4 使用模板產生網頁

- 如前一章看到的，網頁伺服器除了能回傳文字資料，也能分享 **JSON** 這種結構化資料
- 然而，**JSON** 主要的用途是讓程式交換資料，它得解析和處理過才能轉換成適合瀏覽的網頁
- 若網頁的內容格式是有跡可循的，只有資料是動態的，我們可以使用一個新技巧，叫做“網頁模板(**template**)”

- 模板基本上就是一串文字構成的骨架，當中有些部分留白，讓模板引擎抓一些值填進去，如圖所示



- 在上圖的左上角是模板，可以發現{{name}}是填空處
- 當我們嘗試將“John”這個值傳給模板引擎，填空處會被換成那個值，產生出動態內容
- Go語言提供了兩種模板套件，一個用於文字(text/template)，另一個用於HTML(html/template)
- 既然我們在用HTTP伺服器產生網頁，下面就用HTML模板套件，但它操作起來跟文字模板套件是一樣的

html/template能防範跨網站指令攻擊

- html/template和text/template的差異在於，前者會套用自動字跳脫(autoescape), 也就是將符合HTML, CSS, JavaScript 等指令的特殊字元轉換過，以免被用於跨網站指令碼(cross-site scripting, XSS)攻擊
- 舉個例，一個網站會讓使用者輸入姓名，然後直接填入模板的填空處，但這時攻擊者可以故意填入JavaScript碼來使之執行：

```
<script>alert("XSS attack!")</script>
```

- 若模板需要的資料是用URL參數提供，那攻擊者更可藉由提供釣魚網址的方式夾帶程式碼，藉此竊取其他使用者瀏覽網站時填入的個資等等
- 使用html/template套件便能有效防堵這類攻擊

使用html/template套件

- Go語言的HTML模板套件提供一種模板語言，讓我們能像這樣單傳取代填空處的值：

```
{{name}}
```

- 不過，你也能用模板語言進行複雜一點的條件判斷：

```
{{if age}} Hello {{else}} Bye {{end}}
```

上面的意思是若age內容不為nil, 模板引擎就會填入字串Hello, 反之則使用Bye , 條件判斷必須用{{end}}結尾

- 模板變數也不見得只能是簡單的數字或字串，也可以是物件
- 比如我們有個結構，內含一個欄位較ID，你就能像這樣把該欄位填入模板：

`{{.ID}}`

- 這樣非常方便，因為這表示我們只能傳一個結構給模板，而不是傳一堆個別變數，等等就會看到這是如何實現的

練習：套用HTML模板

- 本練習的目的是用模板來打造結構更好的網頁，而其內容是透過URL的QueryString傳入的
- 在以下程式中，會顯示消費者的一些基本資訊：
 - ID(代碼)
 - Name(名字)
 - Surname(姓氏)
 - Age(年齡)

- 因此查詢該網頁時，完整的URL會如下：

`http://localhost:8080/?id=代碼&name=名字&surname=姓氏&age=年齡`

- 為了簡化起見，就算使用者輸入多重參數，程式也只會讀取第一項；若未提供id，那麼起始頁面只會顯示‘資料不存在’；至於其他三項資料，缺少的項目會直接隱藏

```
1 package main
2
3 import (
4     "html/template"
5     "log"
6     "net/http"
7     "strconv"
8 )
9
10 //HTML模板原始字符串
11 var templateStr = `
12 <html>
13     <h1>Customer {{.ID}}</h1>
14     {{if .ID }}
15         <p>Details:</p>
16         <ul>
17             {{if .Name}}<li>Name: {{.Name}}</li>{{end}}
18             {{if .Surname}}<li>Surname: {{.Surname}}</li>{{end}}
19             {{if .Age}}<li>Age: {{.Age}}</li>{{end}}
20         </ul>
21     {{else}}
22         <p>Data not available</p>
23     {{end}}
24 </html>
25 `
26
```



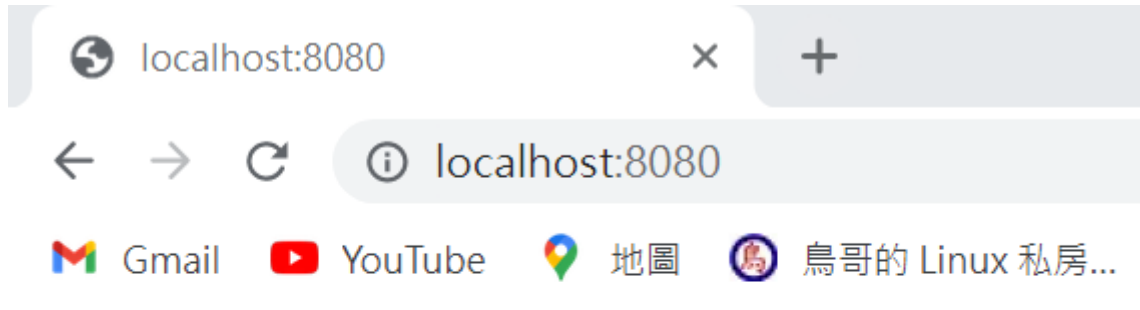
```
27 //要來替模板提供資料的結構
28 type Customer struct {
29     ID      int
30     Name    string
31     Surname string
32     Age     int
33 }
34
35 func hello(w http.ResponseWriter, r *http.Request) {
36     vl := r.URL.Query() //取得查詢參數
37     customer := Customer{}
38
39     id, ok := vl["id"]
40     if ok {
41         customer.ID, _ = strconv.Atoi(id[0])
42     }
43
44     name, ok := vl["name"]
45     if ok {
46         customer.Name = name[0]
47     }
48
49     surname, ok := vl["surname"]
50     if ok {
51         customer.Surname = surname[0]
52     }
53 }
```

```
54     age, ok := vl["age"]
55     if ok {
56         customer.Age, _ = strconv.Atoi(age[0])
57     }
58
59     //建立名為Exercise15.04的模板，並填入templateStr模板字串用於解析
60     tmpl, _ := template.New("Exercise15.04").Parse(templateStr)
61     //使用customer的資料填入模板，並將結果寫入ResponseWriter（傳給客戶端）
62     tmpl.Execute(w, customer)
63 }
64
65 func main() {
66     http.HandleFunc("/", hello)
67     log.Fatal(http.ListenAndServe(":8080", nil))
68 }
69
```

- 模板物件的**Excute()**方法，第一個參數接收**io.Writer**介面型別，而**http.ResponseWriter**就符合這個型別，於是填入值的模板字串(一個**HTML**網頁)就會被傳給客戶端，並在瀏覽器顯示出來
- 以上程式讀取**ID**和**Age**時，呼叫了**strconv.Atoi()**來將字串轉為數字，若轉換出錯，第二個參數會傳回**error**
- 理論上你應該處理錯誤，不過這裡可以忽略，因為**ID**跟年齡輸入錯誤會得到零值，我們也不希望遇到這種錯誤就讓伺服器掛掉

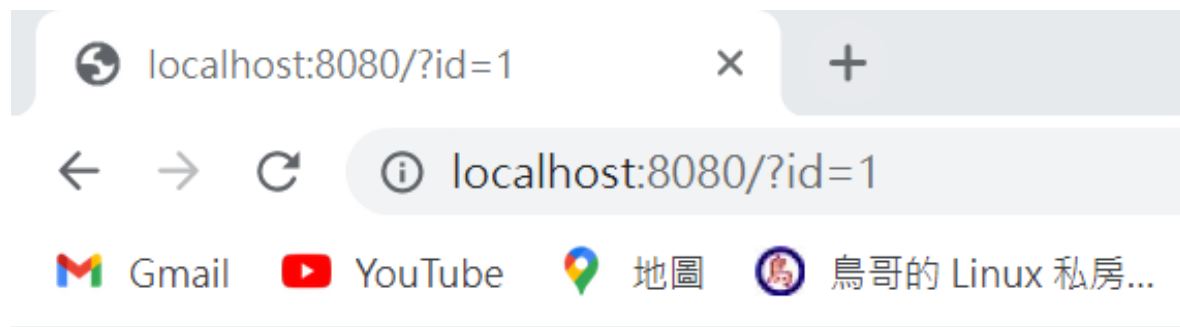
執行程式：

```
PS D:\git\Golang\ch15\15-4> go run .  
[]
```



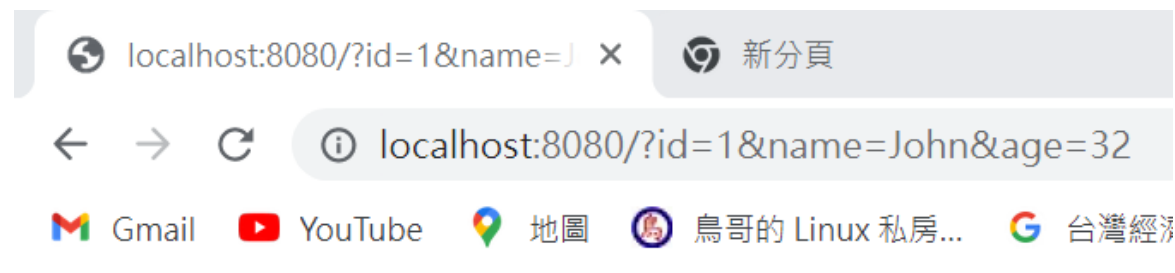
Customer 0

Data not available



Customer 1

Details:



Customer 1

Details:

- Name: John
- Age: 32

- 這個練習展示了如何將網頁的固定部分先定義為模板，然後產生不同網頁
- 但這隻程式的效率還是有點不好，因為每次處理請求都會呼叫 `template.New()` 來產生新的模板
- 更適當的做法是把模板存在一個請求處理器結構中，然後在初始化時產生一次就好了，但這邊為強調模板的用法而將程式簡化了

15-5 使用靜態網頁資源

- 到目前為止，我們的伺服器傳回的東西都是在程式裡先定義好的，但若是想修改回傳訊息就得修改程式碼，重新編譯和重新執行伺服器，這樣不是理想的做法
- 對於這個問題，解法是在伺服器使用靜態檔案，它們會被程式當作外部檔案載入
- 比如前面的模板就可以做成檔案，其他常見的靜態資源還包括網頁圖片/CSS樣式檔/JavaScript指令檔等等

15-5-1 讀取靜態HTML網頁

- 接下來的練習會講解怎麼在伺服器存入特定目錄下的特定靜態檔案，之後也會看到，如何借用靜態的模板檔來產生動態網頁
- 為了把一個靜態檔案當成HTTP回應傳給客戶端，你得在請求處理器的ServeHTTP()方法或請求處理器函式接收的函式內呼叫http.ServeFile()：

```
func ServeFile(w ResponseWriter, r *Request, name string)
```


練習：使用靜態網頁的Hello World 伺服器

- 在這個練習中又要來實作 Hello World 伺服器，但這回改用靜態HTML網頁做為輸出
- 我們也只需要使用一個請求處理函式，使用於該伺服器的所有URL路徑
- 首先，現在練習專案資料夾下建立一個HTML文字檔index.html, 內容很簡單：

ch15 > 15-5-1 > <> index.html >  html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  |   <meta charset="UTF-8">
5  |   <title>Welcome</title>
6  </head>
7  <body>
8  |   <h1>Hello World</h1>
9  </body>
10 </html>
```

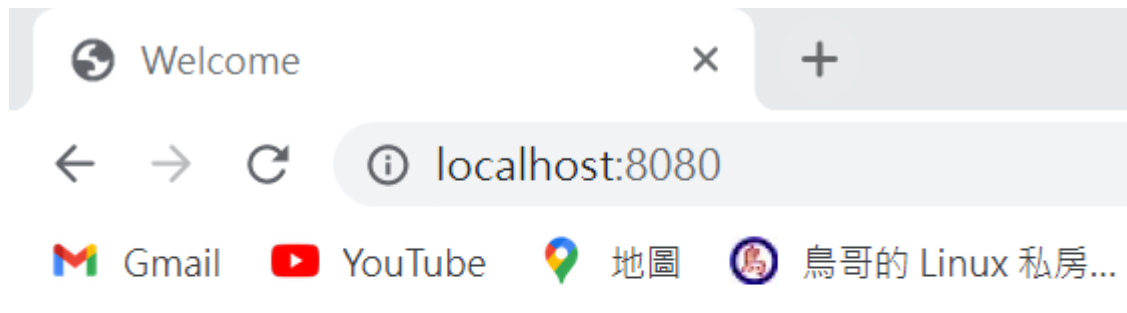
- 接著撰寫main.go程式：

```
1  package main
2
3  import (
4      "log"
5      "net/http"
6  )
7
8  func main() {
9      //直接將一個匿名函式傳給HandleFunc
10     http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
11         //把index.html當成回應寫入ResponseWriter
12         http.ServeFile(w, r, "./index.html")
13     })
14
15     log.Fatal(http.ListenAndServe(":8080", nil))
16 }
17
```

- 接著執行程式：

```
PS D:\git\Golang\ch15\15-5-1> go run .  
[]
```

- 打開 <http://localhost:8080/>，你會看到：

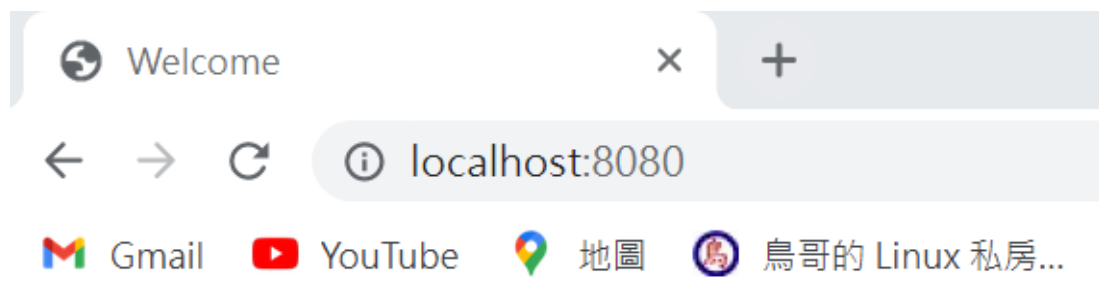


Hello World

- 現在，在不關閉伺服器的情況下修改index.html，將第8行改成：

```
8      <h1>Hello Galaxy</h1>
```

- 然後重新整理瀏覽器：



Hello Galaxy

- 可以看到，當我們把靜態資源和伺服器程式邏輯切割開時，就能在不關閉伺服器的情況下重整網頁了

15-5-2 在伺服器上提供多重靜態資源

- 現在知道如何讓使用者存取一個靜態網頁，你也許會想將這招套用在許多檔案上，甚至寫一個請求處理器結構，把各個檔明用變數的形式存在裡面
- 不過當網頁數量很多時，這樣做就不太實際了
- 甚至，網站的靜態資源不只有html檔，還會包含JS檔，CSS檔，圖片等等，這裡不會詳細探討如何撰寫HTML/CSS/JavaScript，但你仍應該了解該如何從伺服器讓網站和使用者存取這些檔案
- 在以下練習中，我們會用幾個CSS檔來示範

- 從伺服器提供靜態檔案存取，並將模板分散在不同外部檔案中，通常是將專案問題切割成不同區塊的好辦法，使專案容易管理，因此你應該在你所有的網站專案應用這一點
- 若要在HTML網頁加入CSS樣式檔，可以在<head></head>之間加入這個標籤：

`<link rel="stylesheet" href="myfile.css">`

- 這會把名將myfile.css的CSS檔嵌入HTML網頁，並套用該CSS指定的樣式

- 你也已經看過怎麼將檔案系統中的檔案以一對一方式傳回，但若
要傳回的檔案很多，go語言提供了一個更方便的函式：

```
http.FileServer(http.Dir("./public"))
```

- **http.FileServer()**的意義就和字面上的一樣，會開一個檔案伺服器，
而檔案所在的資料夾則用**http.Dir()**取得，上面的例子中，這會讓伺
服器的/public子資料夾下的檔案都能被外界存取，例如：

```
http://localhost:8080/public/myfile.css
```

- 不過在真實世界中，你可能不想讓外界看到伺服器所在機器的目錄位置，因此，你可以讓檔案伺服器對外提供一個不同的路徑：

```
http.StripPrefix("/statics", http.FileServer(http.Dir("./public")))
```

- `StripPrefix()`會將請求檔案的URL當中的“statics”置換成“./public”，並連同檔名傳給檔案伺服器，它會在 ./public 尋找這個檔案
- 換言之，這些檔案從外界看來，就好像至於伺服器的/statics/路徑下



練習：對網頁和使用提供CSS檔



- 在本練習中，你要展示一個歡迎網頁，這些網頁會引用一些外部CSS檔，而這些檔案將透過檔案伺服器來提供
- 此外，這些檔案在伺服器本地的檔案系統位於/**public**資料夾下，但在伺服器上得透過/**statics**路徑存取
- 首先先建立HTML網頁，並用<link...>參照到3個CSS檔：


ch15 > 15-5-2 > <> index.html >  html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Welcome</title>
6      <link rel="stylesheet" href="/statics/body.css">
7      <link rel="stylesheet" href="/statics/header.css">
8      <link rel="stylesheet" href="/statics/text.css">
9  </head>
10 <body>
11     <h1>Hello World</h1>
12     <p>May I give you a warm welcome</p>
13 </body>
14 </html>
```

- 接著建立/public子資料夾，並撰寫以下三個CSS檔：

```
ch15 > 15-5-2 > public > # body.css >  body
1  body {
2      background-color:  beige;
3  }
```

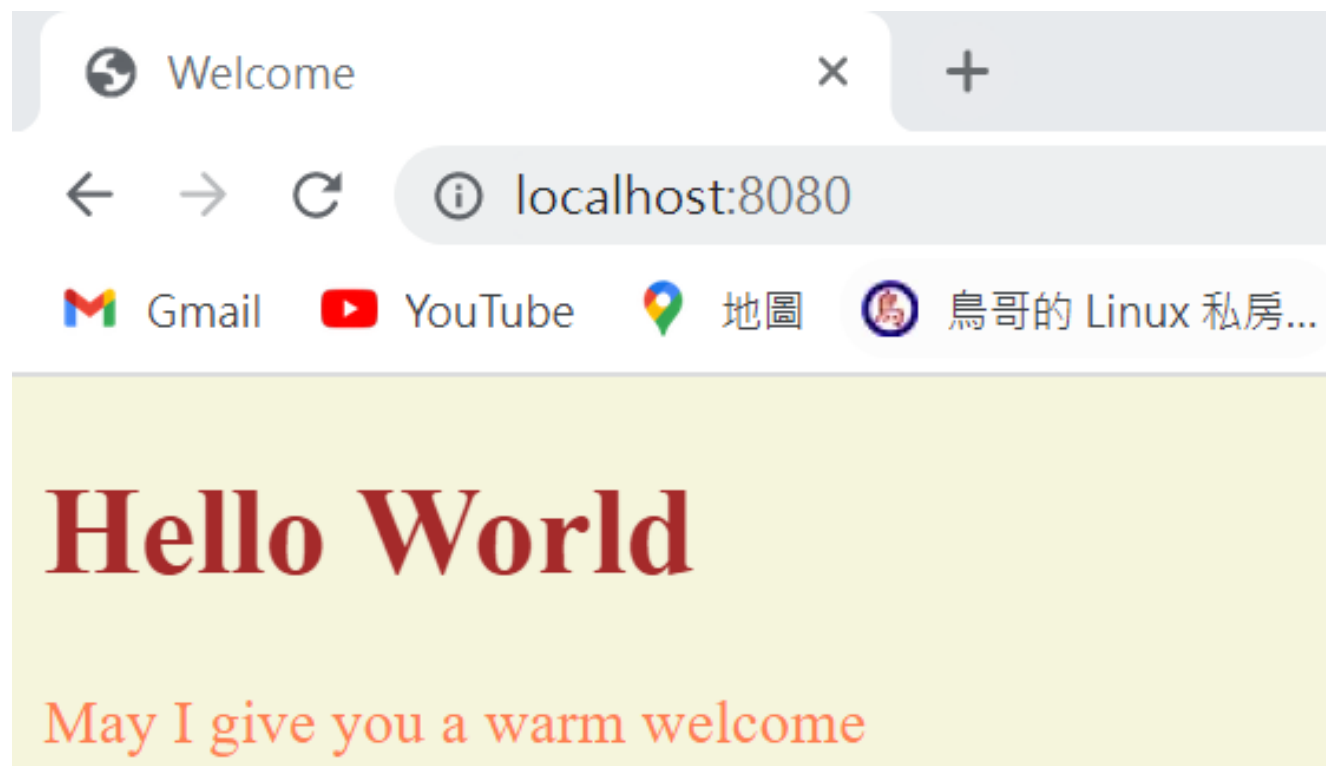
```
ch15 > 15-5-2 > public > # header.css >  h1
1  h1 {
2      color:  brown;
3  }
```

```
ch15 > 15-5-2 > public > # text.css >  p
1  p {
2      color:  coral;
3  }
```

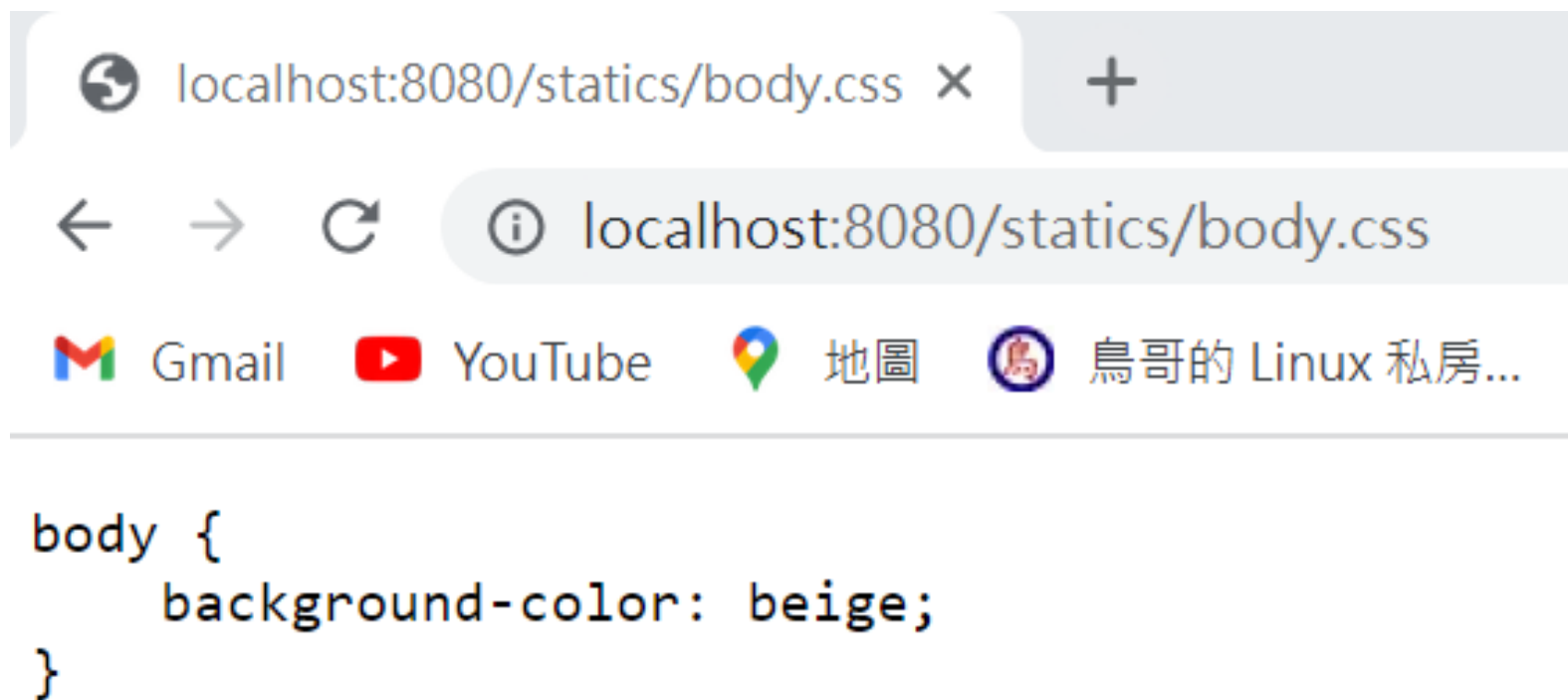
- 最後是專案根目錄下的伺服器程式main.go：

```
1  package main
2
3  import (
4      "log"
5      "net/http"
6  )
7
8  func main() {
9
10     //對任何路徑提供index.html
11     http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
12         http.ServeFile(w, r, "./index.html")
13     })
14     //將/statics路徑對應到本地的/public資料夾
15     http.Handle(
16         "/statics/",
17         http.StripPrefix("/statics/", http.FileServer(http.Dir("./public"))),
18     )
19
20     log.Fatal(http.ListenAndServe(":8080", nil))
21 }
22
```

- 現在啟動伺服器：



- 你會發現，HTML檔確實得透過/statics路徑來存取CSS檔：



15-5-3 使用模板檔案產生動態網頁

- 通常網頁靜態資源會原封不動供人存取，但若你想傳回動態內容網頁，也可以使用外部模板檔案，這讓你能在不重啟伺服器的情況下修改模板
- 不過就算go語言式速度相當快的語言，檔案系統的操作通常都很慢；為提升效能，你可選擇在伺服器啟動時就載入模板，並把它儲存在請求處理器結構中：若很重視網頁存取速度，特別是考量會有多重客戶端連線，便可以考慮這麼做
- 只是這麼做，若你想修改模板，就得重開伺服器，除非使用下一章的併行性運算繞開這個限制，但這裡不回討論這部分


- 應該還記得，前面拿標準的go語言模板來產生動態網頁，現在我們要把模板變成外部資源，在一個HTML檔案裡寫模板語言，然後從伺服器載入它
- 模板引擎會解讀之，把參數填入填空處；對於這個動作，我們可以使用html/template函式：

Func ParseFiles(filenamees ...string) (*Template, error)

- 你可以像這樣呼叫它：`tpl, err := template.ParseFiles("mytemplate.html")`
- 接著再呼叫`tpl.Execute()`就能產生內容了

練習：使用外部模板檔案

- 首先建立外部模板檔案/template/hello_tmpl.html：

```
ch15 > 15-5-3 > template > <> hello_tmpl.html >  html
1  <html>
2  <h1>Customer {{.ID}}</h1>
3  {{if .ID }}
4  <p>Details:</p>
5  <ul>
6  |   {{if .Name}}<li>Name: {{.Name}}</li>{{end}}
7  |   {{if .Surname}}<li>Surname: {{.Surname}}</li>{{end}}
8  |   {{if .Age}}<li>Age: {{.Age}}</li>{{end}}
9  </ul>
10 {{else}}
11 <p>Data not available</p>
12 {{end}}
13 </html>
```

- 在主程式中，改用一個請求處理器結構，並用欄位`tmpl`來記錄模板檔案內容：

```
1  package main
2
3  import (
4      "html/template"
5      "log"
6      "net/http"
7      "strconv"
8  )
9
10 type Customer struct {
11     ID      int
12     Name    string
13     Surname string
14     Age     int
15 }
16
17 //會記錄模板的請求處理器
18 type Hello struct {
19     tmpl *template.Template
20 }
21
```

```
22 //請求處理器方法
23 func (h Hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
24     v1 := r.URL.Query()
25     customer := Customer{}
26
27     id, ok := v1["id"]
28     if ok {
29         customer.ID, _ = strconv.Atoi(id[0])
30     }
31
32     name, ok := v1["name"]
33     if ok {
34         customer.Name = name[0]
35     }
36
37     surname, ok := v1["surname"]
38     if ok {
39         customer.Surname = surname[0]
40     }
41
42     age, ok := v1["age"]
43     if ok {
44         customer.Age, _ = strconv.Atoi(age[0])
45     }
46
47     h.tpl.Execute(w, customer) //使用請求處理器的模板產生動態網頁
48 }
49
```

```
50 func main() {  
51     //建立請求處理  
52     hello := Hello{}  
53     //載入模板檔案和建立模板物件，賦予給請求處理器  
54     hello.tpl, _ = template.ParseFiles("./template/hello_tmpl.html")  
55     //註冊請求處理器  
56     http.Handle("/", hello)  
57     //啟動伺服器  
58     log.Fatal(http.ListenAndServe(":8080", nil))  
59 }  
60
```

15-6 用表單更新POST方法更新伺服器資料

- 到目前為止，你跟前述程式練習的互動都是透過網頁瀏覽器，用**GET**方法取得網頁形式的回應；當然，網頁瀏覽器也能送出**POST**請求，通常是用來上船表單資料
- 在下個練習中，你將看到如何藉由**POST**方法實作一個網頁表單提交系統
- 將來你可以運用更精巧的第三方函式庫，好讓程式更精簡，不過這裡只是展示基礎，且你會發現go語言標準函式庫已經有很多幫助了

練習：問卷填寫表單

- 在這個練習會用到兩個網頁，一個是讓使用者填寫資料用的表單(form)，另一個是用來顯示提交表單的結果
- 首先是問卷畫面form.html, 它包含了<form></form>標籤，三個輸入欄位以及一個送出紐：

ch15 > 15-6 > <> form.html > ...

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Form</title>
6  </head>
7  <h1>Form</h1>
8  <body>
9      <form method="post" action="/hello">
10         <ul>
11             <li>Name: <input type="text" name="name"></li>
12             <li>Surname: <input type="text" name="surname"></li>
13             <li>Age: <input type="text" name="age"></li>
14         </ul>
15         <input type="submit" name="send" value="Send">
16     </form>
17 </body>
18 </html>
```

- 這裡就不深入探討HTML的語法了，但簡單來說，當使用者按下表單內的送出按鈕時，這個表單會對伺服器的/hello路徑送出POST請求，這時表單內所有的標籤的值會包含在請求主體中，並以name屬性來識別不同欄位(name/surname/age)
- 接著用來送出資料後顯示結果的網頁result.html, 其實就是一個模板檔案：

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  |   <meta charset="UTF-8">
5  |   <title>Welcome</title>
6  </head>
7  <body>
8  |   <h1>Details</h1>
9  |   <ul>
10 |       <li>Name: {{.Name}}</li>
11 |       <li>Surname: {{.Surname}}</li>
12 |       <li>Age: {{.Age}}</li>
13 |   </ul>
14 </body>
15 </html>
```

- 最後是主程式main.go, 他會在收到POST請求後讀取表單內容：

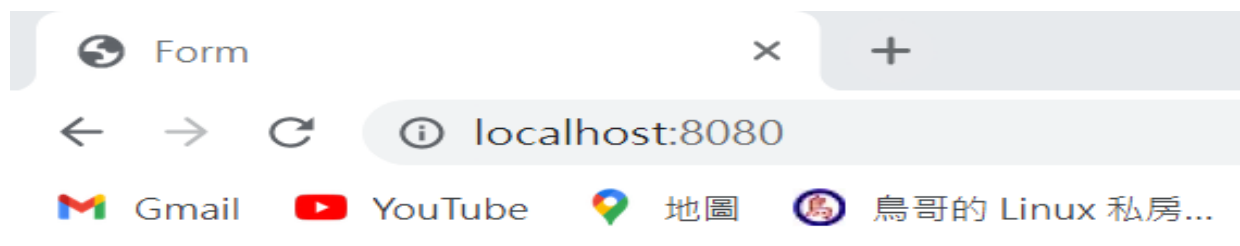
```
1  package main
2
3  ∨ import (
4      |     "html/template"
5      |     "log"
6      |     "net/http"
7  )
8
9  ∨ type Visitor struct { //用來整理使用者以表單送出的資料
10     |     Name    string
11     |     Surname string
12     |     Age     string
13 }
14
15 ∨ type Hello struct { //請求處理器
16     |     tmpl *template.Template //紀錄模板物件的屬性
17 }
18
```

```
19 //請求處理器方法(請求/hello路徑時)
20 func (h Hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
21     visitor := Visitor{}
22
23     //檢查是不是POST請求，不是就回傳HTTP狀態碼405 (method not allowed)
24     if r.Method != http.MethodPost { // 也可以寫成 r.Method != "POST"
25         w.WriteHeader(http.StatusMethodNotAllowed)
26         return
27     }
28
29     err := r.ParseForm()
30     if err != nil {
31         //沒有表單或解讀錯誤的話，回傳HTTP狀態碼400(bad request)
32         w.WriteHeader(http.StatusBadRequest)
33         return
34     }
35
36     //從表單讀取值
37     visitor.Name = r.Form.Get("name")
38     visitor.Surname = r.Form.Get("surname")
39     visitor.Age = r.Form.Get("age")
40
41     h.tmpl.Execute(w, visitor)
42 }
43
```

```
44 //用來設定並傳回請求處理器的函式
45 func NewHello(tmp1Path string) (*Hello, error) {
46     //設定請求處理器要使用的模板
47     tmp1, err := template.ParseFiles(tmp1Path)
48     if err != nil {
49         return nil, err
50     }
51
52     return &Hello{tmp1}, nil
53 }
54
55 func main() {
56     hello, err := NewHello("./result.html") //取得請求處理器
57     if err != nil {
58         log.Fatal(err)
59     }
60     http.Handle("/hello", hello) //要hello處理 /hello(表單) 路徑的請求
61
62     //伺服器根路徑則指向form.html表單畫面
63     http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
64         http.ServeFile(w, r, "./form.html")
65     })
66
67     log.Fatal(http.ListenAndServe(":8080", nil))
68 }
69
```

執行程式：

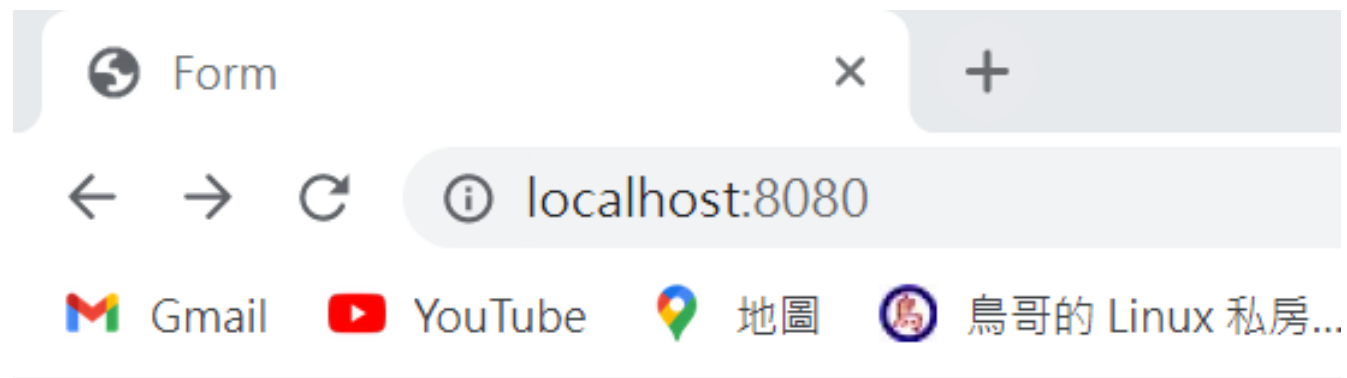
```
PS D:\git\Golang\ch15\15-6> go run .  
[
```



Form

- Name:
- Surname:
- Age:

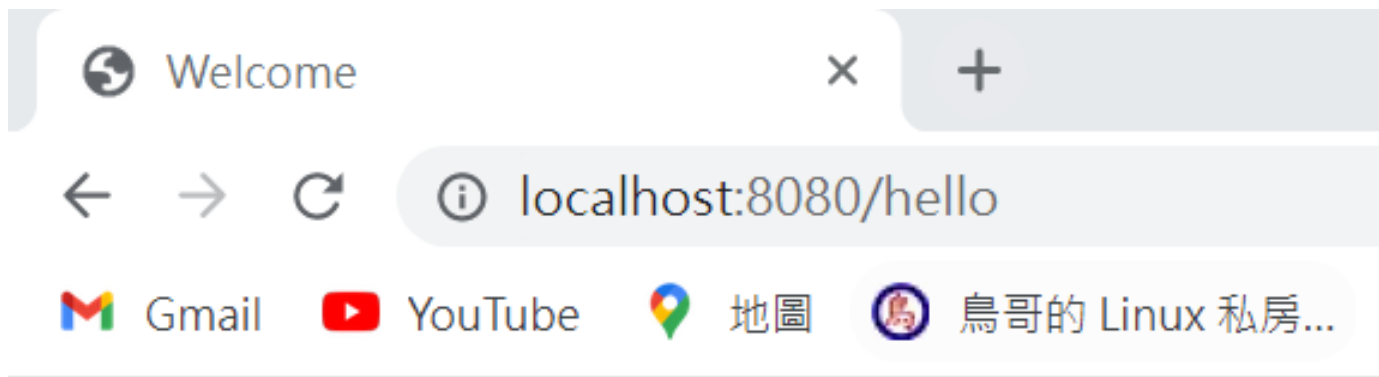
- 填寫資料：



Form

- Name:
- Surname:
- Age:

- 按下send, 跳轉到/hello路徑, 伺服器也用result.html模板顯示剛才填寫的資料：



Details

- Name: John
- Surname: Smith
- Age: 40

15-7 簡易RESTful API：交換JSON資料

- HTTP伺服器不見得都是讓伺服器和人使用，很多時候會用在程式間的相互溝通，而這時通常會用到四海皆準的通用格式，也就是JSON
- REST(Representational State Transfer, 表現層狀態轉換)或RESTful是一種網路API的架構風格，借用了HTTP的幾種請求方法來讓客戶端和伺服器交換資料，常被用於所謂的微服務

- 在HTTP協定中，除了GET和POST以外，還有兩種方法叫做PUT和DELETE
- RESTful服務會用GET來查詢資源，並用POST上傳資源；PUT的用途是更新資源，DELETE則是刪除它
- 當然RESTful是一種軟體風格而非規範，如何使用HTTP方法由你決定，但你仍可以參考RESTful的做法來開發API，讓使用者用更一致的方法操作這些遠端服務

- 這裡不會深入討論如何開發RESTful API, 但各位目前的所學, 例如JSON資料處理/檔案和資料庫存取等, 都足以讓你開發出實用的網路API

練習：能傳回指定時區時間的RESTful API

- 在以下練習中，我們要寫的伺服器會接收使用者的**GET**請求，當中可能包含**URL**參數，然後將對應的時間資料(**UTC**時間/該時區時間/時區名稱)以**JSON**格式傳回
- 你用瀏覽器會看到**JSON**字串，你也許能像第**14**章那樣寫個客戶端測試它，不過這裡會使用一個**API**客戶端程式來測試

```
1  package main
2
3  import (
4      "encoding/json"
5      "log"
6      "net/http"
7      "strings"
8      "time"
9  )
10
11  type WorldTime struct { //對應要回傳的JSON資料結構
12      UTC      string `json:utc`
13      Local    string `json:local`
14      Timezone string `json:timezone`
15  }
16
```

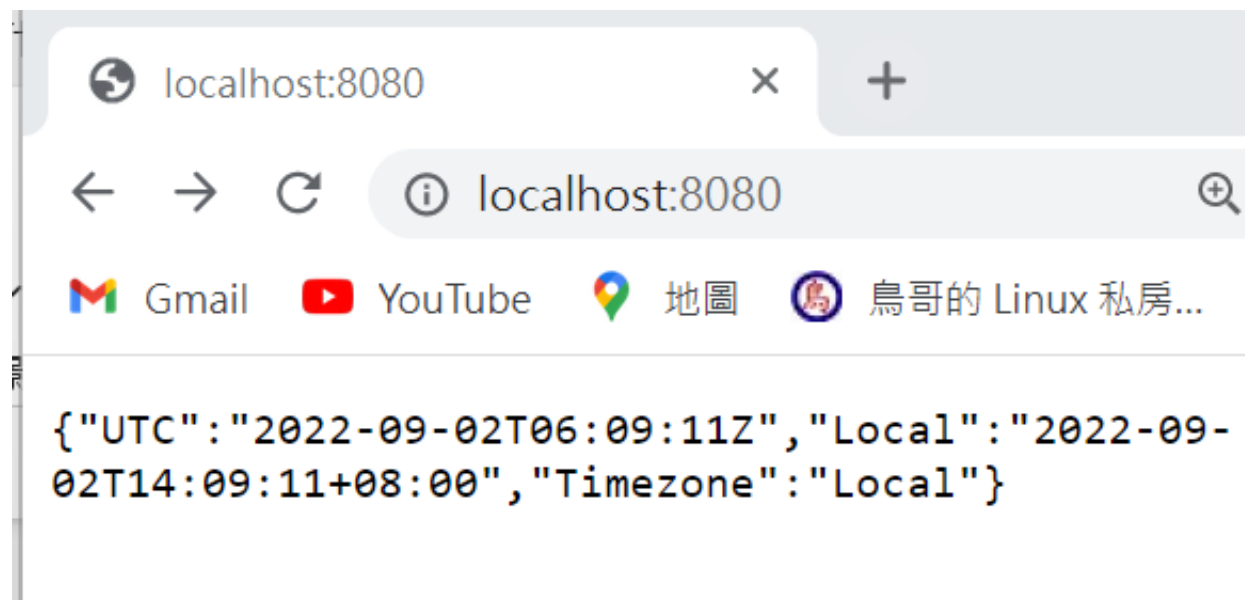
```
17  ✓ func RestfulService(w http.ResponseWriter, r *http.Request) {
18  ✓     if r.Method != http.MethodGet { //若不是GET方法就回傳HTTP 405
19      |         w.WriteHeader(http.StatusMethodNotAllowed)
20      |         return
21      |     }
22
23     w.Header().Set("Content-Type", "application/json")
24     now := time.Now() //取得目前時間
25
26     //讀取參數tz
27     v1 := r.URL.Query()
28     tz, ok := v1["tz"]
29  ✓     if ok {
30      |         //有tz參數的話，嘗試用它來設定時區
31      |         location, err := time.LoadLocation(strings.TrimSpace(tz[0]))
32  ✓     |         if err != nil { //時區錯誤，傳回HTTP 404
33      |         |             w.WriteHeader(http.StatusBadRequest)
34      |         |             //傳回一個時間欄位為空字串，時區則帶有錯誤資訊的JSON資料
35      |         |             jsonBytes, _ := json.Marshal(WorldTime{Timezone: "Invalid timezone name"})
36      |         |             w.Write(jsonBytes)
37      |         |             return
38      |         |         }
39      |         now = now.In(location)
40     }
41 }
```

```
42 //取得要傳回的時間和時區字串
43 worldTime := WorldTime{}
44 worldTime.UTC = now.UTC().Format(time.RFC3339)
45 worldTime.Local = now.Format(time.RFC3339)
46 worldTime.Timezone = now.Location().String()
47
48 jsonBytes, _ := json.Marshal(worldTime)
49 w.Write(jsonBytes)
50 // json.NewEncoder(w).Encode(worldTime)
51 }
52
53 func main() {
54     http.HandleFunc("/", RestfulService)
55     log.Fatal(http.ListenAndServe(":8080", nil))
56 }
57
```


執行程式：

```
PS D:\git\Golang\ch15\15-7> go run .  
[]
```

- 在瀏覽器打開下列網址，先不輸入參數看看：

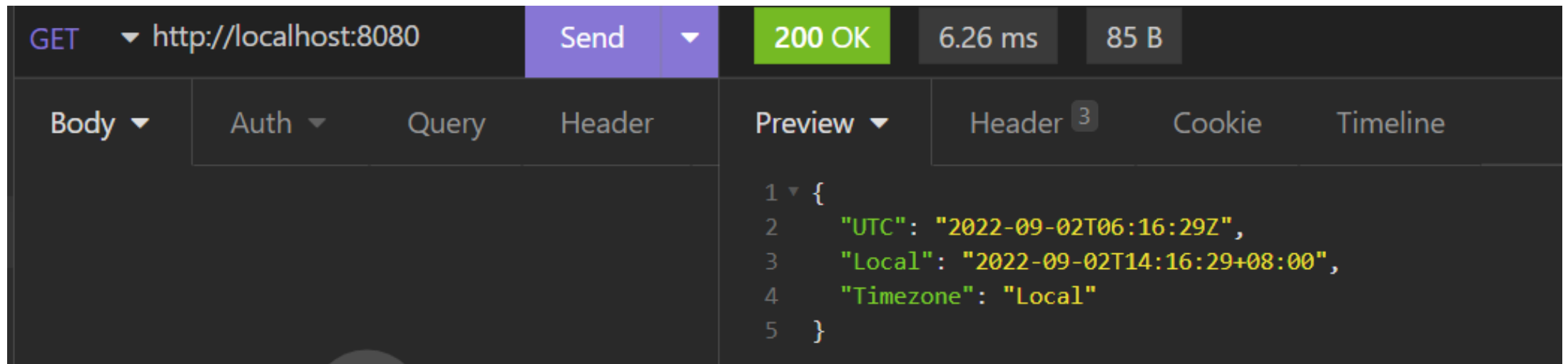


- 不過，既然本練習是讓程式交換資料用的RESTful API, 你也許會想用Insomnia 或 Postman 之類的JSON客戶端測試它：

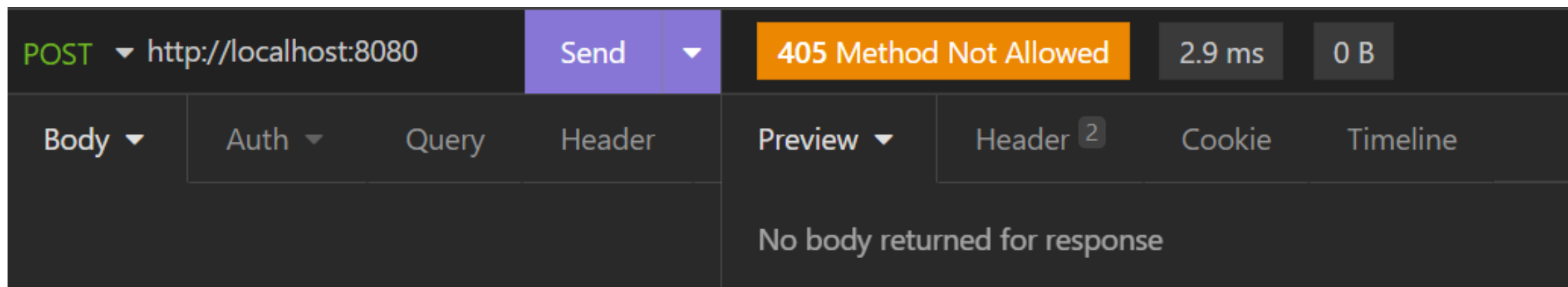
使用Insomnia Core

- 接著到 <https://insomnia.rest/download/> 下載免費的 Insomnia Core
- 啟動後在Dashboard點選Insomnia, 應該會看到用來輸入測試網址與資料的畫面

- 在Insomnia中間上方輸入目標網址(<http://localhost:8080>), 並確保左邊使用的是GET方法, 然後按Send送出, 應該會看到右邊出現伺服器的回應:



- 由於我們並未在網址提供tz參數，因此API沒有設定時區，於是傳回伺服器的本地時間
- 若改成使用POST請求，就會收到HTTP狀態碼405：



- 現在來嘗試加上tz參數，指定時區為Asia/Tokyo：

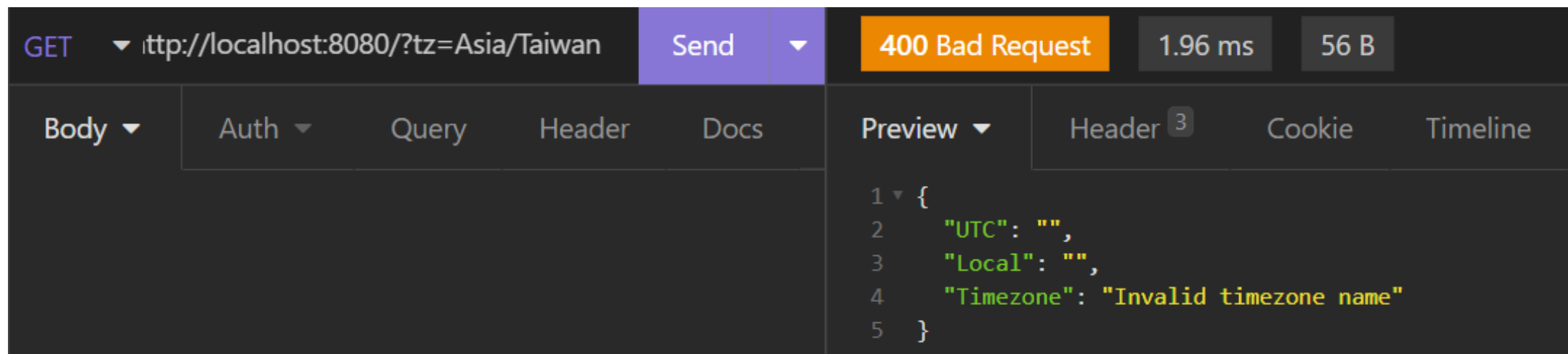
The screenshot shows a web browser's developer tools interface. The top bar displays the request method as GET, the URL as http://localhost:8080/?tz=Asia/Tokyo, and the status as 200 OK. The response time is 30.4 ms and the size is 90 B. The 'Send' button is highlighted. Below the top bar, the 'Body' tab is selected, showing the response body as a JSON object. The JSON object contains three properties: 'UTC' with the value '2022-09-02T06:20:06Z', 'Local' with the value '2022-09-02T15:20:06+09:00', and 'Timezone' with the value 'Asia/Tokyo'.

Method	URL	Status	Time	Size
GET	http://localhost:8080/?tz=Asia/Tokyo	200 OK	30.4 ms	90 B

Tab	Content
Body	<pre>{ "UTC": "2022-09-02T06:20:06Z", "Local": "2022-09-02T15:20:06+09:00", "Timezone": "Asia/Tokyo" }</pre>
Auth	
Query	
Header	
Docs	

Tab	Content
Preview	<pre>1 { 2 "UTC": "2022-09-02T06:20:06Z", 3 "Local": "2022-09-02T15:20:06+09:00", 4 "Timezone": "Asia/Tokyo" 5 }</pre>
Header	3
Cookie	
Timeline	

- 若輸入一個錯誤或不存在的時區，API會傳回 HTTP 狀態碼 404，以及一個時間欄位為空，時區則有錯誤資訊的JSON資料



- 以上的簡易RESTful API 只會接收GET請求，不過你依然能用同樣原理擴充到 POST, PUT 與 DELETE 方法，讓使用者能透過API增刪修改資料

本章結束