

# Cookie & Session



# Cookie & Session

`session` 和 `cookie` 是網站瀏覽中較為常見的兩個概念，也是比較難以辨析的兩個概念，但它們在瀏覽需要認證的服務頁面以及頁面統計中卻相當關鍵。先來了解一下 `session` 和 `cookie` 怎麼來的？考慮這樣一個問題：如何抓取一個存取受限的網頁？如新浪微博好友的主頁，個人微博頁面等。顯然，透過瀏覽器，可以手動輸入使用者名稱和密碼來存取頁面，而所謂的"抓取"，其實就是使用程式來模擬完成同樣的工作，因此需要了解"登入"過程中到底發生了什麼。

當用戶來到微博登入頁面，輸入使用者名稱和密碼之後點選"登入"後瀏覽器將認證資訊 `POST` 給遠端的伺服器，伺服器執行驗證邏輯，如果驗證透過，則瀏覽器會跳轉到登入使用者的微博首頁，在登入成功後，伺服器如何驗證對其他受限制頁面的存取呢？因為 `HTTP` 協議是無狀態的，所以很顯然伺服器不可能知道已經在上一次的 `HTTP` 請求中通過了驗證。當然，最簡單的解決方案就是所有的請求裡面都帶上使用者名稱和密碼，這樣雖然可行，但大大加重了伺服器的負擔(對於每個 `request` 都需要到資料庫驗證)，也大大降低了使用者體驗(每個頁面都需要重新輸入使用者名稱密碼，每個頁面都帶有登入表單)。

既然直接在請求中帶上使用者名稱與密碼不可行，那麼就只有在伺服器或客戶端儲存一些類似的可以代表身份的資訊了，所以就有了 `cookie` 與 `session`。

# Cookie & Session

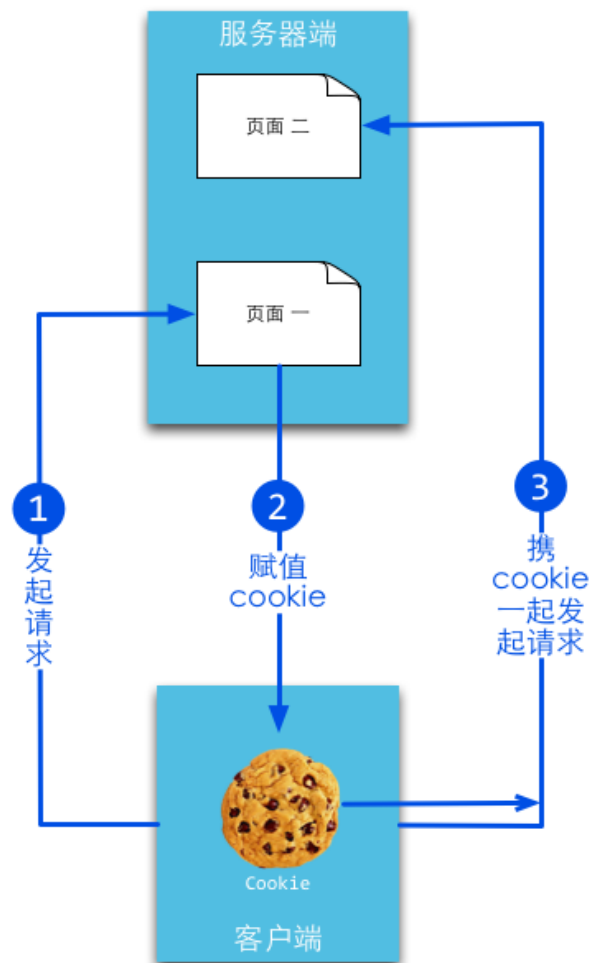
## Cookie

- 簡而言之就是在**本地電腦**儲存一些使用者操作的歷史資訊(當然包括登入資訊)，並在使用者再次存取該站點時瀏覽器透過 HTTP 協議將本地 **cookie** 內容傳送給伺服器，從而完成驗證，或繼續上一步操作。

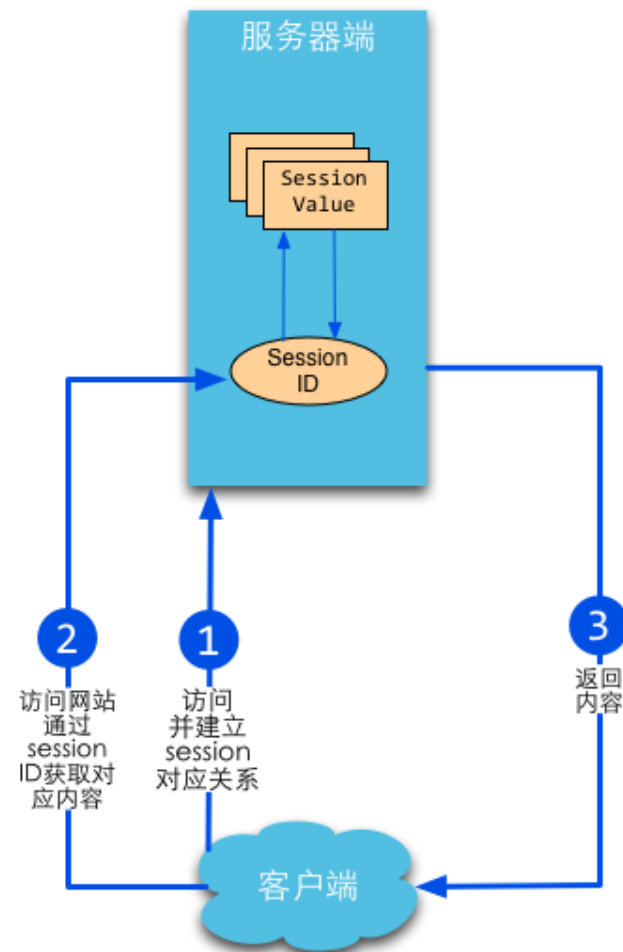
## Session

- 簡而言之就是在**伺服器**上儲存使用者操作的歷史資訊。**伺服器使用 session id 來標識 session**，**session id** 由伺服器負責產生，保證隨機性與唯一性，相當於一個隨機金鑰，避免在握手或傳輸中暴露使用者真實密碼。但該方式下，仍然需要將傳送請求的客戶端與 **session** 進行對應，所以可以藉助 **cookie** 機制來取得客戶端的標識(即 **session id**)，也可以透過 **GET** 方式將 **id** 提交給伺服器。

# Cookie & Session



cookie 原理圖



session 原理圖

# Cookie & Session

## Go 設定 cookie

- 語言中透過 `net/http` 套件中的 `SetCookie` 來設定：

```
http.SetCookie(w ResponseWriter, cookie *Cookie)
```

- `w` 表示需要寫入的 `response`，`cookie` 是一個 `struct`，來看一下 `cookie` 物件是怎麼樣的：

```
type Cookie struct {
    Name      string
    Value      string
    Path       string
    Domain     string
    Expires    time.Time
    RawExpires string

    // MaxAge=0 means no 'Max-Age' attribute specified.
    // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
    // MaxAge>0 means Max-Age attribute present and given in seconds
    MaxAge     int
    Secure     bool
    HttpOnly   bool
    Raw        string
    Unparsed   []string // Raw text of unparsed attribute-value pairs
}
```

# Cookie & Session

```
type Cookie struct {  
    Name string  
    Value string  
    Path string // optional  
    Domain string // optional  
    Expires time.Time // optional  
    RawExpires string // for reading cookies only  
    // MaxAge=0 means no 'Max-Age' attribute specified.  
    // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'  
    // MaxAge>0 means Max-Age attribute present and given in seconds  
    MaxAge int  
    Secure bool  
    HttpOnly bool  
    SameSite SameSite  
    Raw string  
    Unparsed []string // Raw text of unparsed attribute-value pairs  
}
```

- **Name/Value**：Cookie的名稱和值
  - Path：可以將Cookie限定在某個路徑下，只有這個路徑和它的子路徑才可以訪問。
  - Domain：只關聯的web伺服器的Domain，比如example.com。如果設置的Cookie的domain為a.example.com，那麼訪問b.example.com的時候是不能訪問這個Cookie的。
  - Expires：為Cookie過期時間，Cookie超過這個時間點就會被刪除了。
  - RawExpires：Expires字串表示，格式為Wdy, DD Mon YYYY HH:MM:SS或者Wdy, DD Mon YY HH:MM:SS
  - MaxAge：最大存活時間。
  - Secure：設置 Cookie 只在HTTPS的請求中才會作用。
  - **HttpOnly**：跟安全性有關，建議true。
  - SameSite：跟Chrome有關，主要是在處理跨域的安全問題。
- 
- 備註：這邊要特別注意ListenAndServe的位置，如果在HandleFunc之前的話，會導致該路由吃不到噲，所以如果範例跑失敗的話，建議注意一下這個部分
  - HttpOnly：這個部分是建議使用true，當 cookie 有設定HttpOnly true時，瀏覽器會限制cookie只能經由HTTP(S)協定來存取。因此當網站被 XSS 攻擊時，有把 cookie HttpOnly設定為true，這樣一來xss就無法直接透過JavaScript來盜取 cookie。

# Cookie & Session

Cookie與Session是web開發常需要使用的玩意，  
先來個cookie的範例程式：

範例中設定該cookie的name所對應的value，  
以及設定HttpOnly，其實最好還是要設定其存活  
時間也會比較好。

```
package main

import (
    "fmt"
    "net/http"
)

func setCookie(w http.ResponseWriter, r *http.Request) {
    c := http.Cookie{
        Name:    "username",
        Value:   "Tom",
        HttpOnly: true,
    }

    http.SetCookie(w, &c)
}

func getCookie(w http.ResponseWriter, r *http.Request) {
    c, err := r.Cookie("username")
    if err != nil {
        fmt.Fprintln(w, "Cannot get cookie")
    }
    fmt.Fprintln(w, c)
}

func indexHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "hello world")
}

func main() {
    http.HandleFunc("/", indexHandler)
    http.HandleFunc("/cookie/set", setCookie)
    http.HandleFunc("/cookie/get", getCookie)
    http.ListenAndServe("localhost:8000", nil)
}
```

# Cookie & Session

## session 建立過程

- session 的基本原理是由伺服器為每個會話維護一份資訊資料，客戶端和伺服器端依靠一個全域性唯一的標識 (session id) 來存取這份資料，以達到互動的目的。當用戶存取 Web 應用時，伺服器端程式會隨需要建立 session，這個過程可以概括為三個步驟：
  1. 產生全域性唯一識別符號(session id)。
  2. 開闢資料儲存空間。一般會在**記憶體**中建立相應的資料結構，但這種情況下，系統一旦跳電，所有的會話資料就會丟失，如果是電子商務類別網站，這將造成嚴重的後果。所以為了解決這類別問題，可以將會話資料寫到**檔案裡或儲存在資料庫中**，當然這樣會增加 I/O 開銷，但是它可以實現某種程度的 session 持久化，也更有利於 session 的共享。
  3. 將 session 的全域性唯一標示符傳送給客戶端。
- 以上三個步驟中，最關鍵的是如何傳送這個 session 的唯一標識這一步上。考慮到 HTTP 協議的定義，資料無非可以放到請求、表頭或 Body 裡，所以一般來說會有兩種常用的方式：**cookie** 和 **URL 重寫**。



# Cookie & Session

## session 建立過程

- Cookie

- 伺服器端透過設定 `Set-cookie` 表頭就可以將session的識別符號(session id)傳送到客戶端，而客戶端此後的每一次請求都會帶上這個識別符號，另外一般包含session資訊的cookie會將失效時間設定為0(會話 cookie)，即瀏覽器程序有效時間。至於瀏覽器怎麼處理這個0，每個瀏覽器都有自己的方案，但差別都不會太大。

- URL 重寫

- 所謂 URL 重寫，就是在回傳給使用者的頁面裡的所有的 URL 後面追加 session 識別符號(session id)，這樣使用者在收到回應之後，無論點選回應頁面裡的哪個連結或提交表單，都會自動帶上 session 識別符號，從而就實現了會話的保持。雖然這種做法比較麻煩，但是，**如果客戶端禁用了 cookie 的話，此種方案將會是首選。**

# Cookie & Session

## 實現 session 管理

- 透過上面 session 建立過程的講解，應該對 session 有了一個大致上的認識，但是具體到動態頁面技術裡面，又是怎麼實現 session 的呢？下面將結合 session 的生命週期(lifecycle)，來實現 go 語言版本的 session 管理。
- session 管理設計
  - session 管理涉及到如下幾個因素：
    1. 全域性 session 管理器
    2. 保證 session id 的全域性唯一性
    3. 為每個客戶關聯一個 session
    4. session 的儲存(可以儲存到記憶體、檔案、資料庫等)
    5. session 過期處理

# Cookie & Session

實現 session 管理

- Session 管理器

- 定義一個全域性的 session 管理器

```
type Manager struct {  
    cookieName string    // private cookienam  
    lock       sync.Mutex // protects session  
    provider   Provider  
    maxLifeTime int64  
}  
  
func NewManager(providerName, cookieName string, maxLifeTime int64) (*Manager, error) {  
    provider, ok := provides[providerName]  
    if !ok {  
        return nil, fmt.Errorf("session: unknown provide %q (forgotten import?)", providerName)  
    }  
    return &Manager{provider: provider, cookieName: cookieName, maxLifeTime: maxLifeTime}, nil  
}
```

# Cookie & Session

實現 session 管理

- **Session 管理器**

- Go 實現整個的流程應該也是這樣的，在 main 套件中建立一個全域性的 session 管理器：

```
var globalSessions *session.Manager
//然後在 init 函式中初始化
func init() {
    globalSessions, _ = NewManager("memory", "gosessionid", 3600)
}
```

# Cookie & Session

## 實現 session 管理

### ▪ Session 管理器

- session 是儲存在伺服器端的資料，它可以以任何的方式儲存，比如儲存在記憶體、資料庫或者檔案中。

因此抽象出一個 **Provider 介面**，用以表徵 session 管理器底層儲存結構。

```
type Provider interface {  
    SessionInit(sid string) (Session, error)  
    SessionRead(sid string) (Session, error)  
    SessionDestroy(sid string) error  
    SessionGC(maxLifeTime int64)  
}
```

1. SessionInit：實現 Session 的初始化，操作成功則回傳此新的 Session 變數
2. SessionRead：回傳 sid 所代表的 Session 變數，如果不存在，那麼將以 sid 為參數呼叫
3. SessionDestroy：用來刪除 sid 對應的 Session 變數
4. SessionGC：根據 maxLifeTime 來刪除過期的資料

# Cookie & Session

## 實現 session 管理

### ▪ Session 管理器

- 那麼 **Session 介面** 需要實現什麼樣的功能呢？對 Session 的處理基本就**設定值、讀取值、刪除值以及取得當前 sessionID** 這四個操作，所以 Session 介面也就實現這四個操作：

```
type Session interface {  
    Set(key, value interface{}) error // set session value  
    Get(key interface{}) interface{} // get session value  
    Delete(key interface{}) error     // delete session value  
    SessionID() string                // back current sessionID  
}
```

- 以上設計思路來源於 database/sql/driver，先定義好介面，然後具體的儲存 session 的結構實現相應的介面並註冊後，相應功能這樣就可以使用了，以下是用來隨需註冊儲存 session 的結構的 Register 函式的實現。

```
var provides = make(map[string]Provider)  
  
// Register makes a session provide available by the provided name.  
// If Register is called twice with the same name or if driver is nil,  
// it panics.  
func Register(name string, provider Provider) {  
    if provider == nil {  
        panic("session: Register provider is nil")  
    }  
    if _, dup := provides[name]; dup {  
        panic("session: Register called twice for provider " + name)  
    }  
    provides[name] = provider  
}
```

# Cookie & Session

實現 session 管理

- 全域性唯一的 **Session ID**

- Session ID 是用來識別存取 Web 應用的每一個使用者，因此必須保證它是全域性唯一的 (GUID)，

下面程式碼展示了如何滿足這一需求：

```
func (manager *Manager) sessionId() string {  
    b := make([]byte, 32)  
    if _, err := rand.Read(b); err != nil {  
        return ""  
    }  
    return base64.URLEncoding.EncodeToString(b)  
}
```

# Cookie & Session

## 實現 session 管理

- **session 建立**：為每個客戶關聯一個 session
  - 需要為每個來訪使用者分配或取得與他相關連的 Session，以便後面根據 Session 資訊來驗證操作。
  - SessionStart 這個函式就是用來檢測是否已經有某個 Session 與當前來訪使用者發生了關聯，如果沒有則建立之。

```
func (manager *Manager) SessionStart(w http.ResponseWriter, r *http.Request) (session Session) {
    manager.lock.Lock()
    defer manager.lock.Unlock()
    cookie, err := r.Cookie(manager.cookieName)
    if err != nil || cookie.Value == "" {
        sid := manager.sessionId()
        session, _ = manager.provider.SessionInit(sid)
        cookie := http.Cookie{Name: manager.cookieName, Value: url.QueryEscape(sid), Path: "/", HttpOnly: true, MaxAge: int(manager.maxLifeTime)}
        http.SetCookie(w, &cookie)
    } else {
        sid, _ := url.QueryUnescape(cookie.Value)
        session, _ = manager.provider.SessionRead(sid)
    }
    return
}
```



# Cookie & Session

## 實現 session 管理

- session 建立
  - 用login 操作來示範 session 的運用：

```
func login(w http.ResponseWriter, r *http.Request) {  
    sess := globalSessions.SessionStart(w, r)  
    r.ParseForm()  
    if r.Method == "GET" {  
        t, _ := template.ParseFiles("login.gtpl")  
        w.Header().Set("Content-Type", "text/html")  
        t.Execute(w, sess.Get("username"))  
    } else {  
        sess.Set("username", r.Form["username"])  
        http.Redirect(w, r, "/", 302)  
    }  
}
```

# Cookie & Session

## 實現 session 管理

- 操作值：設定、讀取和刪除
- SessionStart 函式回傳的是一個滿足 Session 介面的變數，那麼該如何用他來對 session 資料進行操作呢？上面的例子中的程式碼session.Get("uid")已經展示了基本的讀取資料的操作，現在再來看一下詳細的操作：

```
func count(w http.ResponseWriter, r *http.Request) {
    sess := globalSessions.SessionStart(w, r)
    createtime := sess.Get("createtime")
    if createtime == nil {
        sess.Set("createtime", time.Now().Unix())
    } else if (createtime.(int64) + 360) < (time.Now().Unix()) {
        globalSessions.SessionDestroy(w, r)
        sess = globalSessions.SessionStart(w, r)
    }
    ct := sess.Get("countnum")
    if ct == nil {
        sess.Set("countnum", 1)
    } else {
        sess.Set("countnum", (ct.(int) + 1))
    }
    t, _ := template.ParseFiles("count.gtpl")
    w.Header().Set("Content-Type", "text/html")
    t.Execute(w, sess.Get("countnum"))
}
```

Session 的操作和操作 key/value 資料庫類似：Set、Get、Delete 等操作，因為 Session 有過期的概念，所以定義了 GC 操作，當存取過期時間滿足 GC 的觸發條件後將會引起 GC，但是當進行了任意一個 session 操作，都會對 Session 實體進行更新，都會觸發對最後存取時間的修改，這樣當 GC 的時候就不會誤刪除還在使用的 Session 實體。

# Cookie & Session

## 實現 session 管理

- session 重置
  - Web 應用中有使用者退出這個操作，那麼當用戶退出應用的時候，需要對該使用者的 session 資料進行刪除操作，上面的程式碼已經示範了如何使用 session 重置操作，下面這個函式就是實現了這個功能：

```
//Destroy sessionid
func (manager *Manager) SessionDestroy(w http.ResponseWriter, r *http.Request){
    cookie, err := r.Cookie(manager.cookieName)
    if err != nil || cookie.Value == "" {
        return
    } else {
        manager.lock.Lock()
        defer manager.lock.Unlock()
        manager.provider.SessionDestroy(cookie.Value)
        expiration := time.Now()
        cookie := http.Cookie{Name: manager.cookieName, Path: "/", HttpOnly: true, Expires: expiration, MaxAge: -1}
        http.SetCookie(w, &cookie)
    }
}
```

# Cookie & Session

## 實現 session 管理

- session 刪除
  - Session 管理器如何來管理刪除，只要在 Main 啟動的時候啟動：

```
func init() {  
    go globalSessions.GC()  
}  
  
func (manager *Manager) GC() {  
    manager.lock.Lock()  
    defer manager.lock.Unlock()  
    manager.provider.SessionGC(manager.maxLifeTime)  
    time.AfterFunc(time.Duration(manager.maxLifeTime), func() { manager.GC() })  
}
```

- 可以看到 GC 充分利用了 time 套件中的定時器功能，當超時 maxLifeTime 之後呼叫 GC 函式，這樣就可以保證 maxLifeTime 時間內的 session 都是可用的，類似的方案也可以用於統計線上使用者數之類別的。

# Cookie & Session

## github.com/gorilla/sessions

- 可以使用 `github.com/gorilla/sessions` 套件來輕鬆管理 sessions。 `github.com/gorilla/sessions` 提供兩種 session 管理的方式。

1. 將 session 存在 server 的方式。

2. 將 session 跟著加密的 cookie 存放在 client side(客戶端)的方式。

第2種方式的優點是 server 端不用管理 session 的狀態，server 就算重開，session 也會存在，不用擔心消失。但是缺點也是有的，因為 session 是跟著 cookie stores 走的，所以當資料量大的時候，這種方式是不適合的，以現在 chrome 來說，就有限制 4096 bytes 的大小。

```
PS C:\Users\teacher\Desktop\goProject> go run .\hello.go
hello.go:5:2: no required module provides package github.com/gorilla/sessions; to add it:
  go get github.com/gorilla/sessions
```

```
PS C:\Users\teacher\Desktop\goProject> go get github.com/gorilla/sessions
go: downloading github.com/gorilla/sessions v1.2.1
go: downloading github.com/gorilla/securecookie v1.1.1
go: added github.com/gorilla/securecookie v1.1.1
go: added github.com/gorilla/sessions v1.2.1
```

# Cookie & Session

github.com/gorilla/sessions

- 下面的程式碼實做了三個頁面
  - /home 存取頁面，要登入才會顯示正確訊息
  - /login 模擬登入頁面，存入 session
  - /logout 模擬登出頁面，把狀態修改成未登入

```
package main

import (
    "fmt"
    "github.com/gorilla/sessions"
    "log"
    "net/http"
)

var store *sessions.CookieStore

func init() {
    store = sessions.NewCookieStore([]byte("secret-key"))
}

func main() {
    http.HandleFunc("/home", home)
    http.HandleFunc("/login", login)
    http.HandleFunc("/logout", logout)
    fmt.Println("session server run on port 8080")
    log.Fatal(http.ListenAndServe("localhost:8080", nil))
}

func logout(w http.ResponseWriter, r *http.Request) {
    session, err := store.Get(r, "session-name")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    session.Values["auth"] = nil
    err = session.Save(r, w)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    fmt.Fprintln(w, "logged out.")
}

func login(w http.ResponseWriter, r *http.Request) {
    session, err := store.Get(r, "session-name")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    session.Values["auth"] = true
    err = session.Save(r, w)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    fmt.Fprintln(w, "logged in")
}

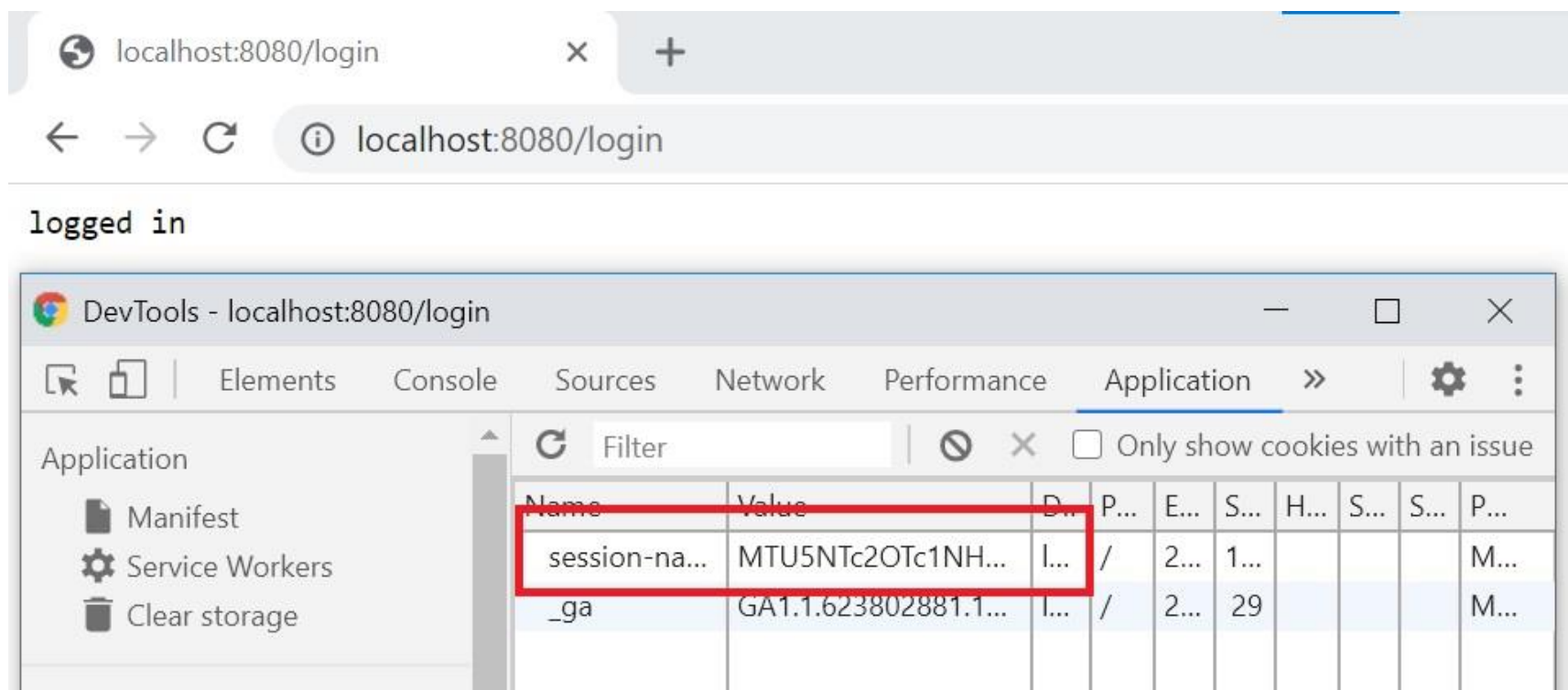
func home(w http.ResponseWriter, r *http.Request) {
    session, err := store.Get(r, "session-name")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    auth := session.Values["auth"]
    if auth != nil {
        isAuth, ok := auth.(bool)
        if ok && isAuth {
            fmt.Fprintln(w, "Home Page")
        } else {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            return
        }
    } else {
        http.Error(w, "unauthorized", http.StatusUnauthorized)
        return
    }
}
```



# Cookie & Session

github.com/gorilla/sessions

- 所以模擬登入login，登入之後，會看到有一個 Cookie Store Sessions 被建立了，且該 cookie 是加密過後的：

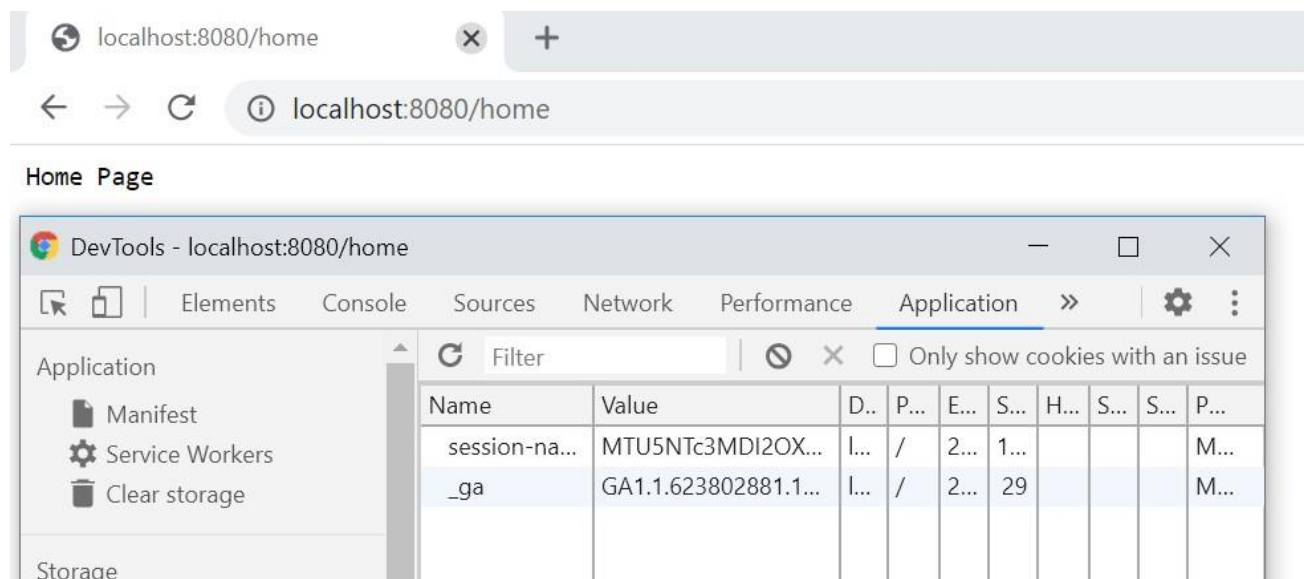




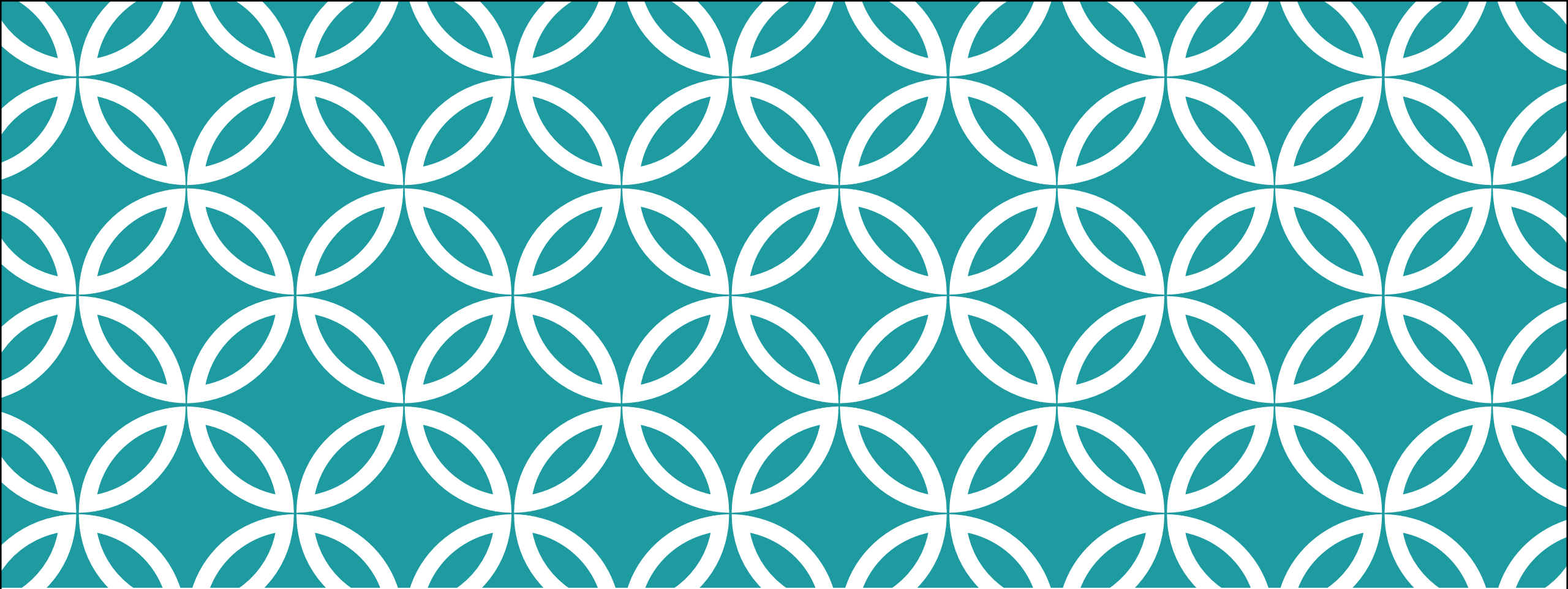
# Cookie & Session

github.com/gorilla/sessions

- 這時候再存取 `/home` 就可以看到有權限的畫面了：



- 最後要注意，`session` 也要加入過期時間，否則認證會一直有效。或者是可以設定 `store.MaxAge()` 設定 `cookie` 存在時間，時間到了 `cookie` 會自行刪除。如果要做 `session key` 的 `rotation`，可以參考 [sessions#NewCookieStore](#)



Go Gin



GO






# Go Gin

前面介紹了 Go 的網路操作套件 `net/http`，那就不能錯過 Go 的網頁框架(`framework`)套件 `gin`，因為要架設一個網站，還是需要考慮到很多情境，例如 `request` 的資料驗證、`middleware`、`response` 的格式，這時就很需要框架來做一個輔助。

## Gin 優勢

- 而為何使用 `gin`，Go 語言又不是只有一種網頁 `framework`，簡單看個表格：

Major Golang Frameworks for Web Development in 2022				
	Name	Stars	Forks	Birth Year
	Gin	25397	2974	2014
	Beego	19401	3992	2012
	Iris	14084	1458	2016
	Echo	13227	1197	2015
	Revel	10855	1307	2011
	Martini	10459	1073	2013
	Buffalo	4172	331	2014



# Go Gin

## Gin 優勢

▪ 這裡大概整理了一下 gin 的優點：

1. Github 上的 star 數最高
2. response 的速度最快、性能好
3. 支援中間層(middleware)
4. 錯誤管理
5. 支援各類型 json xml...等
6. 支援restful
7. CPU 的消耗僅次於 echo
8. 記憶體的表现為最好的

	Golang	Rust	Python	Javascript
type system	static	static	dynamic	dynamic
performance	fast	fast	slow	slow
library	Growing Quickly	Relatively Smaller	Huge	Huge
Interest	High	Most	Low	Medium
Familiarity	Medium	Just started Learning	Most	Low
Framework?	Gin	Rocket	Flask	Empress



# Go Gin

## Gin 安裝

- gin 是在 github 上的套件，而下載套件的方法不只一種，最常用的 go get：

```
go get github.com/gin-gonic/gin
```

- 如此一來，即可安裝套件到 GOPATH 的目錄下，所有的專案都可以使用該套件。





# Go Gin

## Gin 使用

- 直接帶一個範例：

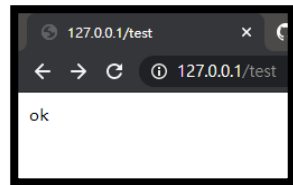
```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    router := gin.Default()
    router.GET("/test", test)
    router.Run(":80")
}

func test(c *gin.Context) {
    str := []byte("ok") // 對於[]byte感到疑惑嗎？ 因為網頁傳輸沒有string的概念，都是要轉成byte字節方式進行傳輸
    c.Data(http.StatusOK, "text/plain", str) // 指定contentType為 text/plain，就是傳輸格式為純文字啦～
}
```

- 先確定80 port沒有被任何程式佔據，否則會出現：listen tcp :80: bind: Only one usage of each socket address (protocol/network address/port) is normally permitted. 的錯誤。
- 從上面的程式碼來看，使用gin的default來建立一個基礎的路由，並把路由的規則跟function都透過他來取得，跟以往的框架差不多，通常這時候框架都會協助封裝，以方便開發者使用，所以可以看到gin.Context的部分，route.run是啟動整個路由來開始為監聽。
- 執行程式後，用瀏覽器打開127.0.0.1/test(本地主機localhost)，成功的話能看到以下畫面：





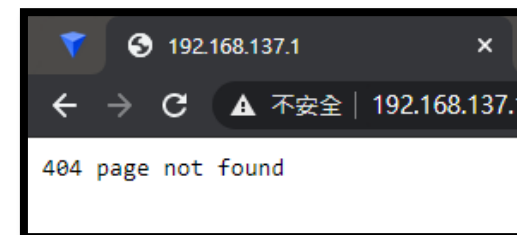
# Go Gin

## Gin 使用

- 回頭看看程式端，方便的地方在於可以：
  - 快速看出Client端的IP
  - 訪問了Server的哪個位置(URI)

```
[GIN-debug] Listening and serving HTTP on :80
[GIN] 2020/09/26 - 23:41:19 | 200 | 975.8µs | 127.0.0.1 | GET | "/test"
[GIN] 2020/09/26 - 23:42:51 | 200 | 0s | 127.0.0.1 | GET | "/test"
[GIN] 2020/09/26 - 23:45:12 | 404 | 0s | 127.0.0.1 | GET | "/announce?info_hash=4r%f9%2c%3d"
[GIN] 2020/09/26 - 23:45:12 | 404 | 0s | 127.0.0.1 | GET | "/announce?info_hash=4r%f9%2c%3d"
[GIN] 2020/09/26 - 23:45:12 | 404 | 0s | 127.0.0.1 | GET | "/announce?info_hash=4r%f9%2c%3d"
[GIN] 2020/09/26 - 23:45:12 | 404 | 0s | 127.0.0.1 | GET | "/announce.php?info_hash=4r%f9%2"
[GIN] 2020/09/26 - 23:45:32 | 404 | 0s | 127.0.0.1 | GET | "/tracker/tracker.php?info_hash="
[GIN] 2020/09/26 - 23:45:48 | 404 | 0s | 127.0.0.1 | GET | "/announce?info_hash=%b9%89%27mY"
[GIN] 2020/09/26 - 23:45:56 | 404 | 0s | 127.0.0.1 | GET | "/tracker/tracker.php?info_hash="
[GIN] 2020/09/26 - 23:47:29 | 404 | 0s | 192.168.137.1 | GET | "/"
[GIN] 2020/09/26 - 23:47:29 | 404 | 0s | 192.168.137.1 | GET | "/favicon.ico"
[GIN] 2020/09/26 - 23:47:32 | 200 | 0s | 192.168.137.1 | GET | "/test"
[GIN] 2020/09/26 - 23:47:53 | 200 | 0s | 192.168.137.142 | GET | "/test"
```

- 由於是開啟/test這個URI，如果Client端訪問根目錄/的話會出現404 page not found：





# Go Gin

什麼是網頁框架呢？

- 就是集大成，讓程式設計師能更快速方便開發網站的一套工具。其實也可以用golang內建原生的函式來達成剛剛的範例的輸出：

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/test", test)
    http.ListenAndServe(":80", nil)
    // 傾聽 TCP 80 port 及 處理服務
}

func test(writer http.ResponseWriter, request *http.Request) {
    // 把字串寫回writer
    str := []byte("ok")
    writer.Write(str)

    // 或者直接用以下這行
    fmt.Fprintf(writer, "ok")
}
```

透過net/http官方的套件以TCP/IP方法來傾聽80 port，

一樣可以跑、得到一樣的結果，而且不用安裝gin套件。

但就沒有那麼方便的儀板表可以來看Client端訪問的資料了。





# Go Gin

## Gin 使用例子

- 寫了兩個函式 `Ready` 和 `Ping`，這是一個很基本的 `health check` 套路，回傳是以 `Json` 格式為主，若程式正常運行，則打到網頁會是回傳函式內的內容。在來要確保 `port` 是沒有其他程式佔據的，在運行程式後，就可以打開自己的瀏覽器：



```
package main

import (
    "fmt"
    "net/http"
    "github.com/gin-gonic/gin"
)

func main() {
    gin.SetMode(gin.DebugMode)
    apiServer := gin.Default()
    apiServer.GET("/ping", Ping)
    apiServer.GET("/ready", Ready)
    err := apiServer.Run(":8787")
    if err != nil {
        fmt.Println("health check:", err)
    }
}

//Ping health check
func Ready(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "status": "UP",
    })
}

func Ping(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "message": "PONG",
    })
}
```



# Go Gin

## Gin 路由

- Gin本身是支援restful的，所以可以看一下底下範例，使用route.就可以直接使用各種HTTP的方式：

```
func main() {  
    // Creates a gin router with default middleware:  
    // logger and recovery (crash-free) middleware  
    router := gin.Default()  
  
    router.GET("/someGet", getting)  
    router.POST("/somePost", posting)  
    router.PUT("/somePut", putting)  
    router.DELETE("/someDelete", deleting)  
    router.PATCH("/somePatch", patching)  
    router.HEAD("/someHead", head)  
    router.OPTIONS("/someOptions", options)  
  
    // By default it serves on :8080 unless a  
    // PORT environment variable was defined.  
    // router.Run()  
    // router.Run(":8080")  
}
```



# Go Gin

## Gin 路由

- 網頁請求方法 HTTP Method
  - HTTP Method 分成了八種GET、POST、DELETE、PUT、HEAD、TRACE、CONNECT、OPTIONS，但只會簡介常用的GET跟POST。
  - GET：
    - 參數都放在網址裡，很不安全(要是參數是登入時的帳號、密碼，就會容易外洩)，但方便開發的時候測試。
  - POST：
    - 參數不在網址裡，比較安全的作法。



# Go Gin

## Gin 路由

- 網頁請求方法 HTTP Method(**restful**)
  - **restful**是一種規範或者風格，並非一種技術或者程式語言，所以嚴格來說不使用不遵守也不會怎樣，但為什麼需要就簡短的說明。  
常見的Http Method有以下：
    - GET：取得資料。
    - POST：新增一筆資料。
    - PUT：更新一筆資料。
    - DELETE：資料刪除。
  - RESTful：在**傳統API**上，可能會看到這樣的function name 或者API name：
    - 新增使用者： /postUser
    - 查所有帳號： /getUsers
    - 查詢使用者： /getUser
    - 修改使用者： /updateUser
    - 刪除使用者： /delUser



# Go Gin

## Gin 路由

- 網頁請求方法 HTTP Method(**restful**)
  - 那如果對方是**使用RESTful**的話：
    - 新增使用者：POST /user
    - 查所有帳號：GET /users
    - 查詢使用者：GET /user/1
    - 修改使用者：PUT /user/1
    - 刪除使用者：DELETE /user/1
  - 從這裡可以看出，RESTful希望操作資料的行為以http method作為依據，所以雙方都可以從http method就可以知道這資料的操作行為是什麼了，以減少傳統在命名方式上，需要心有靈犀一點通的默契。
  - 所以這時候回推一下，一開始的範例，如果是RESTful風格的話，理論上route每一個地都只會對資料做某件事情！例如GET就不應該會更新資料才是。



# Go Gin

## Gin 路由

- Restful(Representational State Transfer)API 是一種設計的概念，其中理念是以資源對應的方式為主，讓每個資源對應Server上的一個URI(以路徑識別資源位置)。
- 不是硬性的特定規範，也沒有明確定義要如何實作，如何達成Restful全賴個人的設計。
- 設計時的大方向：
  - 必要的參數以路徑參數為主，選填的參數以查詢參數為主
  - 靈活運用Method，減少動詞的使用
- 以訊息為例：原先查詢訊息、發布訊息、刪除訊息 3支動作的API，可如下修改，將其命名為相同的func：

```
getMsg    => msg (GET方法)  
createMsg => msg (POST方法)  
deleteMsg => msg (DELETE方法)
```



# Go Gin

## Gin 路由

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

func main() {

    router := gin.Default()

    router.GET("/user/:name", func(c *gin.Context) {
        name := c.Param("name")
        c.String(http.StatusOK, "Hello %s", name)
    })
    router.Run(":8000")
}
```

- 輸入看看`http://localhost:8000/user/Tom`，從程式碼可以猜出：`name`就是對應`c.Param("name")`。
- 當然也可以改成：

```
router.GET("/user/:name/:age", func(c *gin.Context) {
    name := c.Param("name")
    age := c.Param("age")
    c.String(http.StatusOK, "Hello %s", name)
    c.String(http.StatusOK, "age %s", age)
})
```

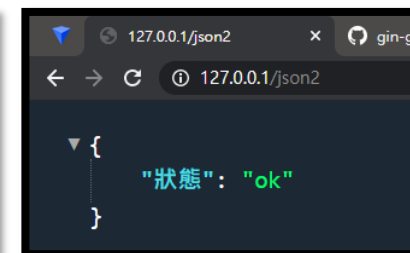
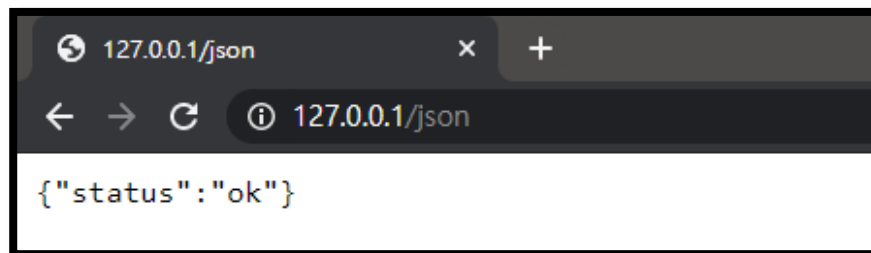


# Go Gin

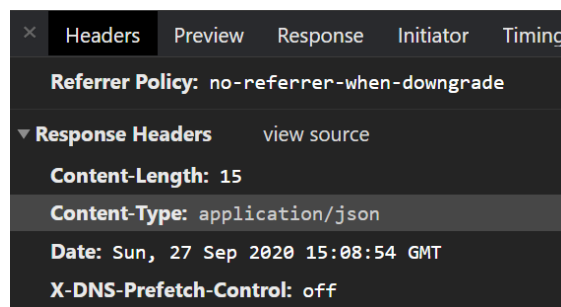
## Gin 路由

- GET 方法 handler
- 以下使用gin不同的兩種方式(Context.Data,Context.JSON)來達成傳回json資料格式的效果：

```
func main() {  
    router := gin.Default()  
  
    router.GET("/json", returnJson)  
    router.GET("/json2", returnJson2)  
  
    router.Run(":8080")  
}  
  
func returnJson(c *gin.Context) {  
    m := map[string]string{"status": "ok"}  
    j, _ := json.Marshal(m)  
    c.Data(http.StatusOK, "application/json", j)  
}  
  
func returnJson2(c *gin.Context) {  
    c.JSON(http.StatusOK, gin.H{  
        "狀態": "ok",  
    })  
}
```



推薦安裝Chrome擴充套件 JSON Viewer Pro，可以在瀏覽器上方便看json檔案的套件。  
另外，可以透過Chrome開發者工具(F12)看到Content-Type為剛剛所設定的







# Go Gin

## Gin 路由

- GET 方法 handler
  - Struct 結構裡的tag :
  - 可以直接在struct中加入json tag，讓gin傳回json格式的時候自動對應轉換。也有支援xml等格式，甚至可以自訂格式：

```
func returnJson3(c *gin.Context) {  
    type Result struct {  
        Status string `json:"status"`  
        Message string `json:"message"`  
    }  
  
    var result = Result{  
        Status: "OK",  
        Message: "This is Json",  
    }  
  
    c.JSON(http.StatusOK, result)  
}
```

記得 struct中的 **Status**、**Message**要字首大寫，要對外給gin套件取用



# Go Gin

## Gin 路由

- GET 接收參數
  - GET Method 是以網址帶參數的方式進行參數傳遞，可分成以下兩種：
    - 查詢參數(Query Params)
    - 路徑參數(Path Params)

傳輸協定      自定義網站名稱      Parameter 參數      Parameter 參數

`https://www.google.com/show/search?q=HelloWorld&k=HelloWorld`

Sub Domain 子網域      網站內容性質識別 (公司或學校)      /path 路徑      /path 路徑



# Go Gin

## Gin 路由

- GET 接收參數

- 查詢參數(Query Params) `127.0.0.1?user=Jack`

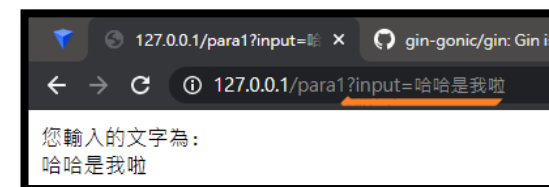
- `127.0.0.1?user=id`：參數名稱為`user`，其中的`id`就是使用者的ID。
    - 為什麼叫作查詢參數或稱作Query String呢？因為使用`?`問號來Query參數，能以這方法夾帶表單參數(HTML Form)，但傳遞表單時使用POST方法更為安全。

```
func main() {
    router := gin.Default()
    router.GET("/para1", para1)

    router.Run(":80")
}

func para1(c *gin.Context) {
    // input := c.DefaultQuery("input", "使用者沒有任何輸入。") // 使用者沒有輸入參數時 可設定預設值
    input := c.Query("input")
    msg := []byte("您輸入的文字為: \n" + input) // 純文字(text/plain)中的換行是\n, 網頁格式(html)中的換行才是<br />
    c.Data(http.StatusOK, "text/plain; charset=utf-8;", msg) // 如果沒有指定文字編碼、拿掉`charset=utf-8;`的話, 中文會變亂碼。
}
```

- 成功執行的話，點開後會出現以下結果：<http://127.0.0.1/para1?input=哈哈是我啦>





# Go Gin

## Gin 路由

- GET 接收參數

- 路徑參數(Path Params) `127.0.0.1/user/Jack`

- `127.0.0.1/user/id`：其中的user是參數名稱，可代表使用者頁面，id就是使用者的ID。

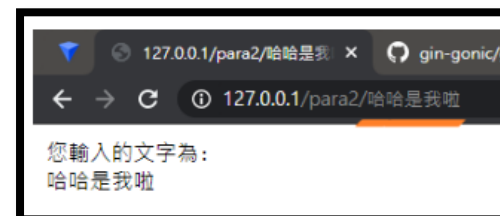
- 路徑即是參數，能以這種方式夾帶參數：

```
func main() {
    router := gin.Default()
    router.GET("/para2/:input", para2)

    router.Run(":80")
}

func para2(c *gin.Context) {
    msg := c.Param("input")
    c.String(http.StatusOK, "您輸入的文字為：\n%s", msg) // 也可使用 `c.String` 返回。第二個參數為組合樣式format
    // c.String(http.StatusOK, msg)                       // 如果沒有組合樣式，可直接輸入字串
}
```

- 成功執行的話，點開後會出現以下結果：<http://127.0.0.1/para2/哈哈是我啦>





# Go Gin

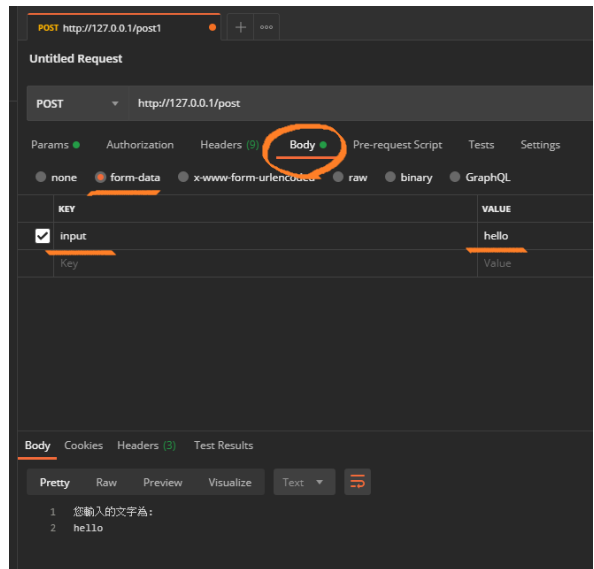
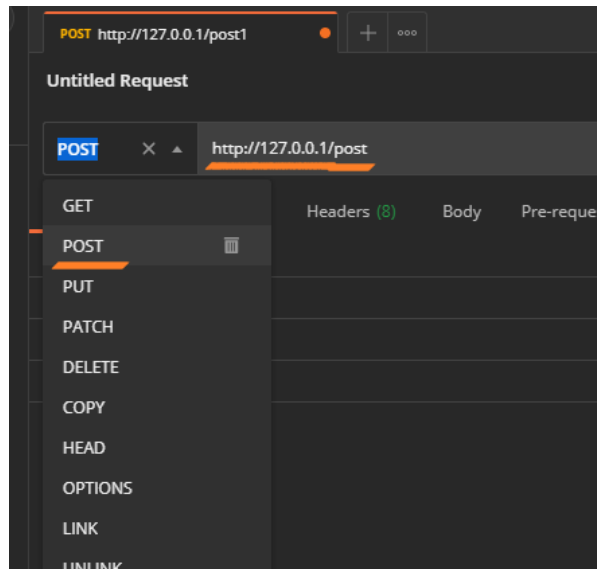
## Gin 路由

- POST 接收參數
- post有寄信、發布訊息的意思：

```
func main() {  
    router := gin.Default()  
    router.RedirectFixedPath = true  
    router.POST("/post", post)  
    router.Run(":80")  
}  
  
func post(c *gin.Context) {  
    //msg := c.PostForm("input")  
    msg := c.DefaultPostForm("input", "表單沒有input。") // 沒有輸入參數時 可設定預設值  
  
    c.String(http.StatusOK, "您輸入的文字為: \n%s", msg)  
}
```

- 由於POST Method參數都隱藏起來了，並非像GET在網址列填入參數就能達到目的，瀏覽器難以測試。在這邊會使用到Postman工具來做測試。測試時選擇POST方法：
  - 選擇Body，格式選擇form-data，並且填入表單的Key、Value：
- gin預設是**\*\*大小寫敏感(case sensitivity)\*\***的，只要對應的URI字符大小寫不同即視為不同，一些使用情境下非常的不便。
- gin 自動修正路徑：加上右邊這行重導向正確的URI，讓大小寫通吃：

```
router.RedirectFixedPath = true
```





# Go Gin

## Gin 路由

- Any
  - router.Any 是任何方法都能夠 handle 的。包含 GET, POST, PUT, PATCH, HEAD, OPTIONS, DELETE, CONNECT, TRACE。

```
func main() {  
    router := gin.Default()  
    router.Any("/any", any)  
  
    router.Run(":80")  
}  
  
func any(c *gin.Context) {  
    c.JSON(http.StatusOK, gin.H{  
        "status": "ok",  
    })  
}
```



# Go Gin

## Gin HTML渲染

- 首先需要在程式碼所在的資料夾下，建立一個view的資料夾，並且在該資料夾下建立一個簡單的html檔案：

```
<html>
  <h1>
    {{ .title }}
  </h1>
</html>
```

- 使用 LoadHTMLGlob() 來讀取 html 所在的位置(所以 LoadHTMLGlob 所指定的資料夾名稱需要跟自己建立的相同)，c.HTML 所指定 hello.html 必須是要跟 html 檔案相呼應。

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

func main() {
    router := gin.Default()
    router.LoadHTMLGlob("view/*")
    router.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "hello.html", gin.H{
            "title": "hello~ html",
        })
    })

    router.Run(":8000")
}
```

也可以有多層結構的做法  
router.LoadHTMLGlob("templates/\*\*/\*.html")



# Go Gin

## Gin HTML渲染

- `route`也可以做多個來使用：

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

func main() {
    router := gin.Default()
    router.LoadHTMLGlob("view/**/*")
    router.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "users/user.html", gin.H{
            "title": "hello",
        })
    })
    router.GET("/users/index2", func(c *gin.Context) {
        c.HTML(http.StatusOK, "users/user.html", gin.H{
            "title": "Users",
        })
    })
    router.Run(":8000")
}
```

但要注意的是如果有多層的話，HTML的內容需要調整一下，必須要加上`define...end`，要特別注意的是開頭跟結尾夾著，這部分一開始沒注意到有結尾的`end`，導致跑不出來。缺一不可：

`{{ define "users/user.html" }}` .... `{{ end }}`

否則會報錯，參考如下：

```
{{ define "users/user.html" }}
<html>
    <h1>
        {{ .title }} user
    </h1>
</html>
{{ end }}
```





# Go Gin

## Gin 表單

- 在gin上該如何使用表單：

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()
    r.POST("/", func(c *gin.Context) {
        str1 := c.PostForm("str1")
        c.String(200, str1)
    })

    r.Run(":8000")
}
```

這部份用指令來跑看看：`curl -d str1=hello http://localhost:8000/`  
可以得到hello，大致上有支援的東西有這些：

<a href="#">Query</a>	<a href="#">PostForm</a>	取得 key 對應的值，若不存在會是空字串
<a href="#">GetQuery</a>	<a href="#">GetPostForm</a>	回傳一個 key 是否存在的結果
<a href="#">QueryArray</a>	<a href="#">PostFormArray</a>	取得 key 對應的陣列，若不存在回傳會是一個空陣列
<a href="#">GetQueryArray</a>	<a href="#">GetPostFormArray</a>	多返回一個 key 是否存在的結果
<a href="#">QueryMap</a>	<a href="#">PostFormMap</a>	獲取 key 對應的 map，若不存在回傳會是空 map
<a href="#">GetQueryMap</a>	<a href="#">GetPostFormMap</a>	回傳一個 key 是否存在的結果
<a href="#">DefaultQuery</a>	<a href="#">DefaultPostForm</a>	若 key 不存在的話，可以指定回傳的預設值

```
fmt.Println(c.PostForm("name"))
fmt.Println(c.PostFormArray("age"))
fmt.Println(c.PostFormMap("sex"))
```



# Go Gin

## Gin Middleware(中間層/中介軟體)

- Middleware 有人翻中間層也有人翻中介軟體，常見的用途之一就是應用在身分驗證的功能，在特定route設定Middleware，此時的Middleware可以想像是進來之前的檢查哨，就不需要在一一自己設置，而是在route那邊處理。(如果是使用vscode，此時可以點Default，就會跳去該程式碼的function)

```
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()
    r.POST("/", func(c *gin.Context) {
        fmt.Println(c.GetPostFormArray("str1"))
    })
    r.Run(":8000")
}
```

```
// Default returns an Engine instance with the Logger and Recovery middleware already attached.
func Default() *Engine {
    debugPrintWARNINGDefault()
    engine := New()
    engine.Use(Logger(), Recovery())
    return engine
}
```

- Default的function如右上：



# Go Gin

## Gin Middleware(中間層/中介軟體)

- 所以可以在不知不覺中，其實在使用gin.Default()返回時，就已經使用了Recovery和Logger這兩個Middleware：

```
package main

import (
    "fmt"
    "time"

    "github.com/gin-gonic/gin"
)

func LoggerHandler() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()
        fmt.Printf("gogo %v \n", t)
    }
}

func main() {
    r := gin.Default()
    r.Use(LoggerHandler())
    //註冊一個Middleware

    r.GET("/", func(c *gin.Context) {
        fmt.Println("HELLO")
    })

    r.Run(":8000")
}
```

這時候可以執行看看<http://localhost:8000/>  
也確實看終端機輸出有print了~



# Go Gin

Gin 實作小專案(製作一個可以提款、存款以及查詢餘額功能的個人小銀行。)

■ 查詢餘額：

```
package main

import (
    "net/http"
    "strconv"

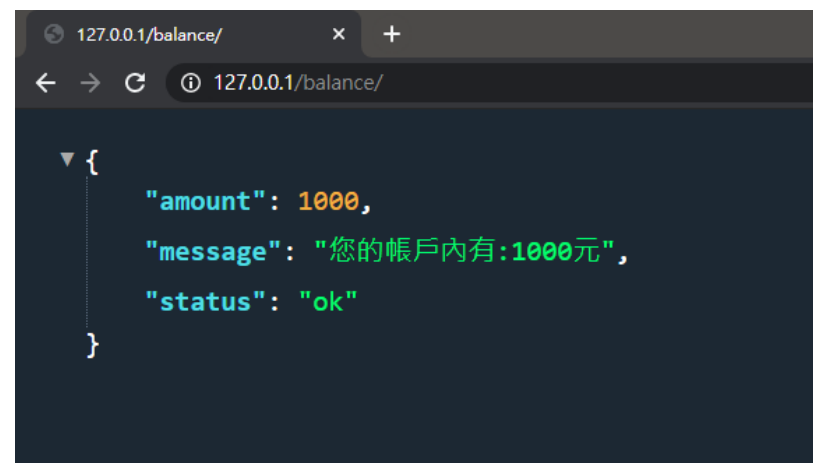
    "github.com/gin-gonic/gin"
)

var balance = 1000

func main() {
    router := gin.Default()
    router.GET("/balance/", getBalance)

    router.Run(":80")
}

func getBalance(context *gin.Context) {
    var msg = "您的帳戶內有:" + strconv.Itoa(balance) + "元"
    context.JSON(http.StatusOK, gin.H{
        "amount": balance,
        "status": "ok",
        "message": msg,
    })
}
```





# Go Gin

Gin 實作小專案(製作一個可以提款、存款以及查詢餘額功能的個人小銀行。)

■ 存款：

```
func deposit(context *gin.Context) {  
    var status string  
    var msg string  
  
    input := context.Param("input")  
    amount, err := strconv.Atoi(input)  
  
    if err == nil {  
        if amount <= 0 {  
            amount = 0  
            status = "failed"  
            msg = "操作失敗，存款金額需大於0元！"  
        } else {  
            balance += amount  
            status = "ok"  
            msg = "已成功存款" + strconv.Itoa(amount) + "元"  
        }  
    } else {  
        amount = 0  
        status = "failed"  
        msg = "操作失敗，輸入有誤！"  
    }  
    context.JSON(http.StatusOK, gin.H{  
        "amount": amount,  
        "status": status,  
        "message": msg,  
    })  
}
```

存款時需要判斷使用者填入的數字是不是正整數，而不是負數或其他亂填的阿雜符號。



# Go Gin

Gin 實作小專案(製作一個可以提款、存款以及查詢餘額功能的個人小銀行。)

- 提款：

```
func withdraw(context *gin.Context) {  
    var status string  
    var msg string  
  
    input := context.Param("input")  
    amount, err := strconv.Atoi(input)  
  
    if err == nil {  
        if amount <= 0 {  
            amount = 0  
            status = "failed"  
            msg = "操作失敗，提款金額需大於0元！"  
        } else {  
            if balance-amount < 0 {  
                amount = 0  
                status = "failed"  
                msg = "操作失敗，餘額不足！"  
            } else {  
                balance -= amount  
                status = "ok"  
                msg = "成功提款" + strconv.Itoa(amount) + "元"  
            }  
        }  
    } else {  
        amount = 0  
        status = "failed"  
        msg = "操作失敗，輸入有誤！"  
    }  
    context.JSON(http.StatusOK, gin.H{  
        "amount": amount,  
        "status": status,  
        "message": msg,  
    })  
}
```

提款時需要判斷使用者填入的數字是不是正整數，  
而不是負數或其他亂填的阿雜符號。



# Go Gin

Gin 實作小專案(製作一個可以提款、存款以及查詢餘額功能的個人小銀行。)

- 賦予API傳回值的意義：

- 與此同時，根據以下發送的參數：<http://127.0.0.1/deposit/100>
- 在操作成功後，就可以從中取得到兩個資訊，
  - 這是儲值操作
  - 並且此筆儲值金額為100
- 相同地，提款也是如此：<http://127.0.0.1/withdraw/10>
- 考量到API的設計理念，可以這樣子改動 API傳回的金額：
- 同時也引入struct作為 gin Context回傳的json結構。

儲值多少錢 => 儲值後用戶餘額有多少  
提款多少錢 => 提款後用戶餘額剩多少

```
type Result struct {  
    Amount int    `json:"amount"`  
    Status string `json:"status"`  
    Message string `json:"message"`  
}  
  
var result = Result{}
```



# Go Gin

Gin 實作小專案(製作一個可以提款、存款以及查詢餘額功能的個人小銀行。)

- 導入回傳值的樣板(**wrapResponse**)：
  - 在設計較大型的專案API時，為了讓每個回傳的json格式、型別都一致，此時可另外設計一個wrapResponse function，不論程式有沒有出現錯誤，都可以將gin.Context 作為參數傳遞給wrapResponse，把所要回傳的值、型別、甚至err都集結起來，統一一個介面來作回傳。如此一來也能更精簡化程式碼。

```
func wrapResponse(context *gin.Context, amount int, err error) {  
    var r = struct {  
        Amount int    `json:"amount"`  
        Status  string `json:"status"`  
        Message string `json:"message"`  
    }{  
        Amount: amount,  
        Status: "ok", // 預設狀態為ok  
        Message: "",  
    }  
  
    if err != nil {  
        r.Amount = 0  
        r.Status = "failed" // 若出現任何err，狀態改為failed  
        r.Message = err.Error() // Message回傳錯誤訊息  
    }  
  
    context.JSON(http.StatusOK, r)  
}
```