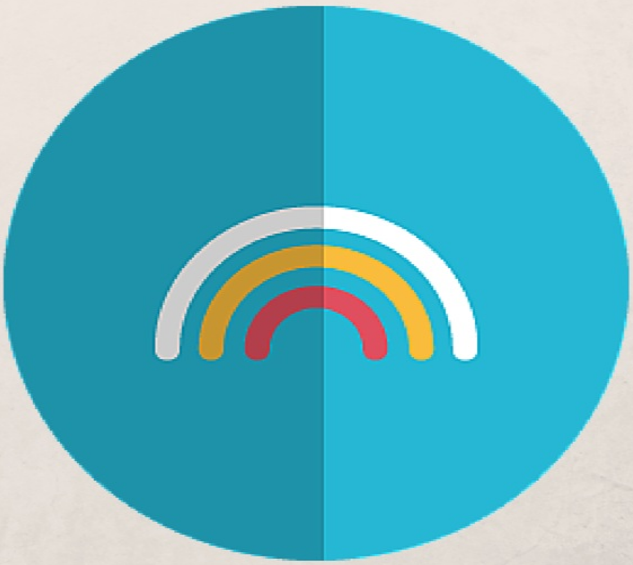
Iris web framework

The complete guide



brought to you absolutely free

github.com/kataras/iris

Table of Contents

PRACTICAL GUIDE [THE DONATOR EDITION]

IRIS WEB FRAMEWORK

BY GERASIMOS MAROPOULOS

#The Fastest Web Framework

Table of Contents

- Context
- Logger
- HTTP access control
- Basic Authentication
- OAuth, OAuth2
- JSON Web Tokens(JWT)
- Secure
- Sessions
- Websockets
- Graceful
- Recovery
- Plugins
- Internationalization and Localization
- Easy Typescript
- Browser based Editor
- Control panel
- Examples

Why

Go is a great technology stack for building scalable, web-based, back-end systems for web

applications.

When you think about building web applications and web APIs, or simply building HTTP servers in Go, does your mind go to the standard net\/http package?

Then you have to deal with some common situations like dynamic routing (a.k.a parameterized), security and authentication, real-time communication and many other issues that net\/http doesn't solve.

The net\/http package is not complete enough to quickly build well-designed back-end web systems. When you realize this, you might be thinking along these lines:

- Ok, the net\/http package doesn't suit me, but there are so many frameworks, which one will work for me?!
- Each one of them tells me that it is the best. I don't know what to do!

The truth

I did some deep research and benchmarks with 'wrk' and 'ab' in order to choose which framework would suit me and my new project. The results, sadly, were really disappointing to me.

I started wondering if golang wasn't as fast on the web as I had read... but, before I let Golang go and continued to develop with nodejs, I told myself:

> '**Makis, don't lose hope, give at least a chance to Golang. Try to build something totally new without basing it off the "slow" code you saw earlier; learn the secrets of this language and make *others* follow your steps!**'.

These are the words I told myself that day [**13 March 2016**].

The same day, later the night, I was reading a book about Greek mythology. I saw an ancient goddess' name and was inspired immediately to give a name to this new web framework (which I had already started writing) - **Iris**.

Two months later, I'm writing this intro.

I'm still here [because Iris has succeed in being the fastest go web framework] (https:\/\/github.com\/kataras\/iris#benchmarks)

Features

- **Switch between template engines**: Select the way you like to parse your html files, switchable via one-line configuration, read more
- **Typescript**: Auto-compile & Watch your client side code via the typescript plugin
- **Online IDE**: Edit & Compile your client side code when you are not home via the editor plugin
- **Iris Online Control**: Web-based interface to control the basics functionalities of your server via the iriscontrol plugin. Note that Iris control is still young
- **Subdomains**: Easy way to express your api via custom and dynamic subdomains*
- **Named Path Parameters**: Probably you already know what this means. If not, It's easy to learn about
- **Custom HTTP Errors**: Define your own html templates or plain messages when http errors occur*
- **Internationalization**: i18n
- **Bindings**: Need a fast way to convert data from body or form into an object? Take a look here
- **Streaming**: You have only one option when streaming comes into play*
- **Middlewares**: Create and/or use global or per route middleware with Iris' simplicity*
- **Sessions**: Sessions provide a secure way to authenticate your clients/users *
- **Realtime**: Realtime is fun when you use websockets*
- **Context**: Context is used for storing route params, storing handlers, sharing variables between middleware, render rich content, send files and much more*
- **Plugins**: You can build your own plugins to inject into the Iris framework*
- **Full API**: All http methods are supported*
- **Party**: Group routes when sharing the same resources or middleware. You can organise a party with domains too! *
- **Transport Layer Security**: Provide privacy and data integrity between your server and the client*
- **Multi server instances**: Not only does Iris have a default main server, you

can declare as many as you need*

- **Zero configuration**: No need to configure anything for typical usage. Well-structured default configurations everywhere, which you can change with ease
- **Zero allocations**: Iris generates zero garbage
- and much more, take a fast look to all sections

Versioning

Current: **v4.0.0-alpha.4**

Read more about Semantic Versioning 2.0.0

- http://semver.org/
- https://en.wikipedia.org/wiki/Software_versioning
- https://wiki.debian.org/UpstreamGuide#Releases_and_Versions

Install

Compatible with go1.6+

```
$ go get -u github.com/kataras/iris/iris
```

this will update the dependencies also.

- If you are connected to the Internet through **China**, according to this you may having problem install Iris. **Follow the below steps**:

1. https://github.com/northbright/Notes/blob/master/Golang/china/get-golang-packages-on-golang-org-in-china.md

1. `$ go get github.com/kataras/iris/iris` **without -u**

- If you have any problems installing Iris, just delete the directory `$GOPATH/src/github.com/kataras/iris` , open your shell and run `go get -u github.com/kataras/iris/iris` .

Hi

```
package main

import "github.com/kataras/iris"

func main() {
    iris.Get("/hi", func(ctx *iris.Context) {
        ctx.Write("Hi %s", "iris")
    })
    iris.Listen(":8080")
}
```

The same

```
package main

import "github.com/kataras/iris"

func main() {
    api := iris.New()
    api.Get("/hi", hi)
    api.Listen(":8080")
}

func hi(ctx *iris.Context){
   ctx.Write("Hi %s", "iris")
}
```

Rich Hi with **html\/template**

```
<!-- ./templates/hi.html -->
<html><head> <title> Hi Iris [THE TITLE] </title> </head>
  <body>
    <h1> Hi {{.Name}} </h1>
  </body>
</html>
```

```
// ./main.go
import "github.com/kataras/iris"

func main() {
    iris.Get("/hi", hi)
    iris.Listen(":8080")
}

func hi(ctx *iris.Context){
   ctx.Render("hi.html", struct { Name string }{ Name: "iris" })
}
```

Rich Hi with **Django-syntax**

```
<!-- ./mytemplates/hi.html -->
<html><head> <title> Hi Iris </title> </head>
  <body>
    <h1> Hi {{ Name }}
  </body>
</html>
```

```
// ./main.go
import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/template/django"
)

func main() {
    iris.UseTemplate(django.New()).Directory("./mytemplates", ".
html")
    iris.Get("/hi", hi)
    iris.Listen(":8080")
}

func hi(ctx *iris.Context){
   ctx.Render("hi.html", map[string]interface{}{"Name": "iris"},
 iris.RenderOptions{"gzip":true})
}
```

- More about render and template engines here

TLS

```
// Listen starts the standalone http server
// which listens to the addr parameter which as the form of
// host:port
//
// It panics on error if you need a func to return an error, use
 the ListenTo
// ex: err := iris.ListenTo(config.Server{ListeningAddr:":8080"})

Listen(addr string)

// ListenTLS Starts a https server with certificates,
// if you use this method the requests of the form of 'http://'
will fail
// only https:// connections are allowed
// which listens to the addr parameter which as the form of
// host:port
//
// It panics on error if you need a func to return an error, use
 the ListenTo
// ex: err := iris.ListenTo(":8080","yourfile.cert","yourfile.ke
y")
ListenTLS(addr string, certFile string, keyFile string)

// ListenUNIX starts the process of listening to the new request
s using a 'socket file', this works only on unix
//
// It panics on error if you need a func to return an error, use
 the ListenTo
// ex: err := iris.ListenTo(":8080", Mode: os.FileMode)
ListenUNIX(addr string, mode os.FileMode)

// ListenVirtual is useful only when you want to test Iris, it d
oesn't starts the server but it configures and returns it
// initializes the whole framework but server doesn't listens to
 a specific net.Listener
```

```
// it is not blocking the app
ListenVirtual(optionalAddr ...string) *Server

// ListenTo listens to a server but acceots the full server's co
nfiguration
// returns an error, you're responsible to handle that
// or use the iris.Must(iris.ListenTo(config.Server{}))
//
// it's a blocking func
ListenTo(cfg config.Server) (err error)

// Close terminates all the registered servers and returns an er
ror if any
// if you want to panic on this error use the iris.Must(iris.Clo
se())
Close() error
```

```
iris.Listen(":8080")
err := iris.ListenTo(config.Server{ListeningAddr: ":8080"})

iris.ListenTLS(":8080", "myCERTfile.cert", "myKEYfile.key")
err := iris.ListenTo(config.Server{ListeningAddr: ":8080", CertF
ile: "myCERTfile.cert", KeyFile: "myKEYfile.key"})
```

Handlers

Handlers should implement the Handler interface:

```
type Handler interface {
    Serve(*Context)
}
```

Using Handlers

```
type myHandlerGet struct {
}

func (m myHandlerGet) Serve(c *iris.Context) {
    c.Write("From %s", c.PathString())
}

//and so on


iris.Handle("GET", "/get", myHandlerGet{})
iris.Handle("POST", "/post", post)
iris.Handle("PUT", "/put", put)
iris.Handle("DELETE", "/delete", del)
```

Using HandlerFuncs

HandlerFuncs should implement the Serve(*Context) func. HandlerFunc is most simple method to register a route or a middleware, but under the hoods it's acts like a Handler. It's implements the Handler interface as well:

```
type HandlerFunc func(*Context)

func (h HandlerFunc) Serve(c *Context) {
    h(c)
}
```

HandlerFuncs shoud have this function signature:

```
func handlerFunc(c *iris.Context)  {
    c.Write("Hello")
}


iris.HandleFunc("GET","/letsgetit",handlerFunc)
//OR
iris.Get("/letsgetit", handlerFunc)
iris.Post("/letspostit", handlerFunc)
iris.Put("/letputit", handlerFunc)
iris.Delete("/letsdeleteit", handlerFunc)
```

Using Handler API

HandlerAPI is any custom struct which has an `*iris.Context` field and known methods signatures.

Before continue I will liked to notice you that this method is slower than `iris.Get, Post..., Handle, HandleFunc` .

I know maybe sounds awful but I, my self not using it, I did it because developers used to use frameworks with the 'MVC' pattern, so think it like the 'C|Controller'. If you don't care about routing performance(~ms) and you like to spent some code time, you're free to use it.

Instead of writing Handlers\/HandlerFuncs for eachone API routes, you can use the `iris.API` function.

```
API(path string, api HandlerAPI, middleware ...HandlerFunc) erro
r
```

For example, for a user API you need some of these routes:

- GET `/users` , for selecting all
- GET `/users/:id` , for selecting specific
- PUT `/users` , for inserting
- POST `/users/:id` , for updating
- DELETE `/users/:id` , for deleting

Normally, with HandlerFuncs you should do something like this:

```
iris.Get("/users", func(ctx *iris.Context){})
iris.Get("/users/:id", func(ctx *iris.Context){ id := ctx.Param(
"id") })

iris.Put("/users",...)

iris.Post("/users/:id", ...)

iris.Delete("/users/:id", ...)
```

But with API you can do this instead:

```
package main

import (
    "github.com/kataras/iris"
)

type UserAPI struct {
    *iris.Context
}

// GET /users
func (u UserAPI) Get() {
    u.Write("Get from /users")
    // u.JSON(iris.StatusOK,myDb.AllUsers())
}

// GET /:param1 which its value passed to the id argument
func (u UserAPI) GetBy(id string) { // id equals to u.Param("par
am1")
    u.Write("Get from /users/%s", id)
    // u.JSON(iris.StatusOK, myDb.GetUserById(id))

}

// PUT /users
func (u UserAPI) Put() {
    name := u.FormValue("name")
```

```
    // myDb.InsertUser(...)
    println(string(name))
    println("Put from /users")
}

// POST /users/:param1
func (u UserAPI) PostBy(id string) {
    name := u.FormValue("name") // you can still use the whole C
ontext's features!
    // myDb.UpdateUser(...)
    println(string(name))
    println("Post from /users/" + id)
}

// DELETE /users/:param1
func (u UserAPI) DeleteBy(id string) {
    // myDb.DeleteUser(id)
    println("Delete from /" + id)
}

func main() {
    iris.API("/users", UserAPI{})
    iris.Listen(":8080")
}
```

As you saw you can still get other request values via the *iris.Context, API has all the flexibility of handler\/handlerfunc.

If you want to use **more than one named parameter**, simply do this:

```
// users/:param1/:param2
func (u UserAPI) GetBy(id string, otherParameter string) {}
```

API receives a third parameter which are the middlewares, is optional parameter:

```
func main() {
    iris.API("/users", UserAPI{}, myUsersMiddleware1, myUsersMid
dleware2)
    iris.Listen(":8080")
}

func myUsersMiddleware1(ctx *iris.Context) {
    println("From users middleware 1 ")
    ctx.Next()
}
func myUsersMiddleware2(ctx *iris.Context) {
    println("From users middleware 2 ")
    ctx.Next()
}
```

Available methods: "GET", "POST", "PUT", "DELETE", "CONNECT", "HEAD", "PATCH", "OPTIONS", "TRACE" should use this **naming conversion**: **Get\/GetBy, Post\/PostBy, Put\/PutBy** and so on...

Using native http.Handler

> Not recommended and I will not help you if any issue comes up, it is just there for your first conversional steps. Note also that using native http handler you cannot access url params.

```
type nativehandler struct {}

func (_ nativehandler) ServeHTTP(res http.ResponseWriter, req *h
ttp.Request) {

}

func main() {
    iris.Handle("", "/path", iris.ToHandler(nativehandler{}))
    //"" means ANY(GET,POST,PUT,DELETE and so on)
}
```

Using native http.Handler via iris.ToHandlerFunc()

```
iris.Get("/letsget", iris.ToHandlerFunc(nativehandler{}))
iris.Post("/letspost", iris.ToHandlerFunc(nativehandler{}))
iris.Put("/letsput", iris.ToHandlerFunc(nativehandler{}))
iris.Delete("/letsdelete", iris.ToHandlerFunc(nativehandler{}))
```

Middleware

Quick view

```
// First point to the static files
iris.Static("/assets", "./public/assets", 1)

// Then declare which middleware to use (custom or not)
iris.Use(myMiddleware{})
iris.UseFunc(func(ctx *iris.Context){})

// Now declare routes
iris.Get("/myroute", func(c *iris.Context) {
    // do stuff
})
iris.Get("/secondroute", myMiddlewareFunc, myRouteHandlerfunc)

// Now run our server
iris.Listen(":8080")


// myMiddleware will be like that

type myMiddleware struct {
  // your 'stateless' fields here
}

func (m *myMiddleware) Serve(ctx *iris.Context){
  // ...
}
```

Middleware in Iris is not complicated, they are similar to simple Handlers. They implement the Handler interface as well:

```
type Handler interface {
    Serve(*Context)
}
type Middleware []Handler
```

Handler middleware example:

```
type myMiddleware struct {}

func (m *myMiddleware) Serve(c *iris.Context){
    shouldContinueToTheNextHandler := true

    if shouldContinueToTheNextHandler {
        c.Next()
    }else{
        c.Text(403,"Forbidden !!")
    }

}

iris.Use(&myMiddleware{})

iris.Get("/home", func (c *iris.Context){
    c.HTML(iris.StatusOK,"<h1>Hello from /home </h1>")
})

iris.Listen(":8080")
```

HandlerFunc middleware example:

```
func myMiddleware(c *iris.Context){
    c.Next()
}

iris.UseFunc(myMiddleware)
```

HandlerFunc middleware for a specific route:

```
func mySecondMiddleware(c *iris.Context){
    c.Next()
}

iris.Get("/dashboard", func(c *iris.Context) {
    loggedIn := true
    if loggedIn {
        c.Next()
    }
}, mySecondMiddleware, func (c *iris.Context){
    c.Write("The last HandlerFunc is the main handler, everythin
g before that is middleware for this route /dashboard")
})

iris.Listen(":8080")
```

> Note that middleware must come before route declaration.

Make use of the middleware, view practical examples here

```
package main

import (
 "github.com/kataras/iris"
 "github.com/iris-contrib/middleware/logger"
)

type Page struct {
    Title string
}

iris.Use(logger.New(iris.Logger))

iris.Get("/", func(c *iris.Context) {
    c.Render("index.html", Page{"My Index Title"})
})

iris.Listen(":8080")
```

API

Use of GET, POST, PUT, DELETE, HEAD, PATCH & OPTIONS

```
package main

import "github.com/kataras/iris"

func main() {
    iris.Get("/home", testGet)
    iris.Post("/login", testPost)
    iris.Put("/add", testPut)
    iris.Delete("/remove", testDelete)
    iris.Head("/testHead", testHead)
    iris.Patch("/testPatch", testPatch)
    iris.Options("/testOptions", testOptions)

    iris.Listen(":8080")
}

func testGet(c *iris.Context) {
    //...
}
func testPost(c *iris.Context) {
    //...
}

//and so on....
```

Declaration

You have wondered this:

- Q: Other frameworks need more lines to start a server, why is Iris different?
- A: Iris gives you the freedom to choose between three ways to use Iris

    1. global **iris.**
    2. declare a new iris station with default config: **iris.New()**
    3. declare a new iris station with custom config: **api := iris.New(config.Iris{...})**

Config can change after declaration with 1&2. `iris.Config.` 3. \/ `api.Config.`

```
import "github.com/kataras/iris"

// 1.
func firstWay() {

    iris.Get("/home",func(c *iris.Context){})
    iris.Listen(":8080")
}
// 2.
func secondWay() {

    api := iris.New()
    api.Get("/home",func(c *iris.Context){})
    api.Listen(":8080")
}
```

Before looking at the 3rd way, let's take a quick look at the **config**.Iris**:

```
type (
    // Iris configs for the station
    Iris struct {
```

```
        // DisablePathCorrection corrects and redirects the requ
ested path to the registed path
        // for example, if /home/ path is requested but no handl
er for this Route found,
        // then the Router checks if /home handler exists, if ye
s,
        // (permant)redirects the client to the correct path /ho
me
        //
        // Default is false
        DisablePathCorrection bool

        // DisablePathEscape when is false then its escapes the
path, the named parameters (if any).
        // Change to true it if you want something like this htt
ps://github.com/kataras/iris/issues/135 to work
        //
        // When do you need to Disable(true) it:
        // accepts parameters with slash '/'
        // Request: http://localhost:8080/details/Project%2FDelta

        // ctx.Param("project") returns the raw named parameter:
 Project%2FDelta
        // which you can escape it manually with net/url:
        // projectName, _ := url.QueryUnescape(c.Param("project"
).
        // Look here: https://github.com/kataras/iris/issues/135
 for more
        //
        // Default is false
        DisablePathEscape bool

        // DisableBanner outputs the iris banner at startup
        //
        // Default is false
        DisableBanner bool

        // ProfilePath a the route path, set it to enable http p
prof tool
        // Default is empty, if you set it to a $path, these rou
```

```
tes will handled:
        // $path/cmdline
        // $path/profile
        // $path/symbol
        // $path/goroutine
        // $path/heap
        // $path/threadcreate
        // $path/pprof/block
        // for example if '/debug/pprof'
        // http://yourdomain:PORT/debug/pprof/
        // http://yourdomain:PORT/debug/pprof/cmdline
        // http://yourdomain:PORT/debug/pprof/profile
        // http://yourdomain:PORT/debug/pprof/symbol
        // http://yourdomain:PORT/debug/pprof/goroutine
        // http://yourdomain:PORT/debug/pprof/heap
        // http://yourdomain:PORT/debug/pprof/threadcreate
        // http://yourdomain:PORT/debug/pprof/pprof/block
        // it can be a subdomain also, for example, if 'debug.'
        // http://debug.yourdomain:PORT/
        // http://debug.yourdomain:PORT/cmdline
        // http://debug.yourdomain:PORT/profile
        // http://debug.yourdomain:PORT/symbol
        // http://debug.yourdomain:PORT/goroutine
        // http://debug.yourdomain:PORT/heap
        // http://debug.yourdomain:PORT/threadcreate
        // http://debug.yourdomain:PORT/pprof/block
        ProfilePath string
        // DisableTemplateEngines set to true to disable loading
 the default template engine (html/template) and disallow the us
e of iris.UseEngine
        // default is false
        DisableTemplateEngines bool
        // IsDevelopment iris will act like a developer, for exa
mple
        // If true then re-builds the templates on each request
        // default is false
        IsDevelopment bool

        // Charset character encoding for various rendering
        // used for templates and the rest of the responses
```

```
        // defaults to "UTF-8"
        Charset string

        // Gzip enables gzip compression on your Render actions,
 this includes any type of render, templates and pure/raw content

        // If you don't want to enable it globaly, you could jus
t use the third parameter on context.Render("myfileOrResponse",
structBinding{}, iris.RenderOptions{"gzip": true})
        // defaults to false
        Gzip bool

        // Sessions contains the configs for sessions
        Sessions Sessions

        // Websocket contains the configs for Websocket's server
 integration
        Websocket *Websocket

        // Tester contains the configs for the test framework, s
o far we have only one because all test framework's configs are
setted by the iris itself
        // You can find example on the https://github.com/katara
s/iris/glob/master/context_test.go
        Tester Tester
    }
)
```

```
// 3.
package main

import (
  "github.com/kataras/iris"
  "github.com/kataras/iris/config"
)

func main() {
    c := config.Iris{
        ProfilePath:        "/mypath/debug",
    }
    // to get the default: c := config.Default()

    api := iris.New(c)
    api.Listen(":8080")
}
```

> Note that with 2. & 3. you **can define and Listen with more than one Iris server** in the same app, when it's necessary.

For profiling there are eight (8) generated routes with pages filled with info:

- \/mypath\/debug\/
- \/mypath\/debug\/cmdline
- \/mypath\/debug\/profile
- \/mypath\/debug\/symbol
- \/mypath\/debug\/goroutine
- \/mypath\/debug\/heap
- \/mypath\/debug\/threadcreate
- \/mypath\/debug\/pprof\/block

- More about configuration here

Configuration

Configuration is a relative package `github.com/kataras/iris/config`

> No need to download it separately, it's downloaded automatically when you install Iris.

Why?

I took this decision after a lot of thought and I ensure you that this is the best and easiest architecture:

- change the configs without needing to re-write all of their fields.

```
irisConfig := config.Iris{ DisablePathCorrection: true }
api := iris.New(irisConfig)
```

- **easy to remember**: `iris` type takes `config.Iris`, sessions takes `config.Sessions`, `iris.Config.Render` is of type `config.Render`, `iris.Config.Render.Template` is the type `config.Template`, `Logger` takes `config.Logger` and so on...
- **easy to search & find out what features exists and what you can change**: just navigate to the config folder and open the type you want to learn about, for example `/websocket.go /iris.Websocket 's` configuration is inside `/config/websocket.go`
- Enables you to do this **without setting up a config yourself**: `iris.Config.Gzip = true` or `iris.Config.Charset = "UTF-8"`.
- **(Advanced usage) merge configs**:

```
//...
import "github.com/kataras/iris/config"
//...
websocketFromDefault:= config.DefaultWebsocket()
//....
websocketManual:= config.Websocket{  Endpoint: "/ws"}

websocketConfig := websocketFromDefault.MergeSingle(websocketMan
ual)
```

Click here to view all station's configs

Party

Let's party with Iris web framework!

```
package main

import "github.com/kataras/iris"

func main() {
    admin := iris.Party("/admin", func(ctx *iris.Context){ ctx.W
rite("Middleware for all party's routes!") })
    {
        // add a silly middleware
        admin.UseFunc(func(c *iris.Context) {
            //your authentication logic here...
            println("from ", c.PathString())
            authorized := true
            if authorized {
                c.Next()
            } else {
                c.Text(401, c.PathString()+" is not authorized f
or you")
            }

        })
        admin.Get("/", func(c *iris.Context) {
            c.Write("from /admin/ or /admin if you pathcorrectio
n on")
        })
        admin.Get("/dashboard", func(c *iris.Context) {
            c.Write("/admin/dashboard")
        })
        admin.Delete("/delete/:userId", func(c *iris.Context) {
            c.Write("admin/delete/%s", c.Param("userId"))
        })
    }
```

```
    beta := admin.Party("/beta")
    beta.Get("/hey", func(c *iris.Context) { c.Write("hey from /
admin/beta/hey") })

    iris.Listen(":8080")
}
```

Subdomains

Subdomains are split into two categories, first is the static subdomain and second is the dynamic subdomain.

- static : when you know the subdomain, usage: `controlpanel.mydomain.com`
- dynamic : when you don't know the subdomain, usage: `user1993.mydomain.com` , `otheruser.mydomain.com`

Iris has the simplest known form for subdomains, simple as Parties.

Static

```
package main

import (
    "github.com/kataras/iris"
)

func main() {
    api := iris.New()

    // first the subdomains.
    admin := api.Party("admin.")
    {
        // admin.mydomain.com
        admin.Get("/", func(c *iris.Context) {
            c.Write("INDEX FROM admin.mydomain.com")
        })
        // admin.mydomain.com/hey
        admin.Get("/hey", func(c *iris.Context) {
            c.Write("HEY FROM admin.mydomain.com/hey")
        })
        // admin.mydomain.com/hey2
        admin.Get("/hey2", func(c *iris.Context) {
            c.Write("HEY SECOND FROM admin.mydomain.com/hey")
        })
    }

    // mydomain.com/
    api.Get("/", func(c *iris.Context) {
        c.Write("INDEX FROM no-subdomain hey")
    })

    // mydomain.com/hey
    api.Get("/hey", func(c *iris.Context) {
        c.Write("HEY FROM no-subdomain hey")
    })

    api.Listen("mydomain.com:80")
}
```

Dynamic\/Wildcard

```
// Package main an example on how to catch dynamic subdomains -
wildcard.
// On the first example (subdomains_1) we saw how to create rout
es for static subdomains, subdomains you know that you will have.

// Here we will see an example how to catch unknown subdomains,
dynamic subdomains, like username.mydomain.com:8080.

package main

import "github.com/kataras/iris"

// register a dynamic-wildcard subdomain to your server machine(
dns/...) first, check ./hosts if you use windows.
// run this file and try to redirect: http://username1.mydomain.
com:8080/ , http://username2.mydomain.com:8080/ , http://usernam
e1.mydomain.com/something, http://username1.mydomain.com/somethi
ng/sadsadsa

func main() {
    /* Keep note that you can use both of domains now (after 3.0
.0-rc.1)
       admin.mydomain.com,  and for other the Party(*.) but this
 is not this example's purpose

    admin := iris.Party("admin.")
    {
        // admin.mydomain.com
        admin.Get("/", func(c *iris.Context) {
            c.Write("INDEX FROM admin.mydomain.com")
        })
        // admin.mydomain.com/hey
        admin.Get("/hey", func(c *iris.Context) {
            c.Write("HEY FROM admin.mydomain.com/hey")
        })
        // admin.mydomain.com/hey2
        admin.Get("/hey2", func(c *iris.Context) {
```

```
            c.Write("HEY SECOND FROM admin.mydomain.com/hey")
        })
    }*/

    dynamicSubdomains := iris.Party("*.")
    {
        dynamicSubdomains.Get("/", dynamicSubdomainHandler)

        dynamicSubdomains.Get("/something", dynamicSubdomainHand
ler)

        dynamicSubdomains.Get("/something/:param1", dynamicSubdo
mainHandlerWithParam)
    }

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Write("Hello from mydomain.com path: %s", ctx.PathSt
ring())
    })

    iris.Get("/hello", func(ctx *iris.Context) {
        ctx.Write("Hello from mydomain.com path: %s", ctx.PathSt
ring())
    })

    iris.Listen("mydomain.com:8080")
}

func dynamicSubdomainHandler(ctx *iris.Context) {
    username := ctx.Subdomain()
    ctx.Write("Hello from dynamic subdomain path: %s, here you c
an handle the route for dynamic subdomains, handle the user: %s"
, ctx.PathString(), username)
    // if  http://username4.mydomain.com:8080/ prints:
    // Hello from dynamic subdomain path: /, here you can handle
 the route for dynamic subdomains, handle the user: username4
}

func dynamicSubdomainHandlerWithParam(ctx *iris.Context) {
    username := ctx.Subdomain()
```

```
    ctx.Write("Hello from dynamic subdomain path: %s, here you c
an handle the route for dynamic subdomains, handle the user: %s"
, ctx.PathString(), username)
    ctx.Write("THE PARAM1 is: %s", ctx.Param("param1"))
}
```

> You can still set unlimitted number of middleware\/handlers to the dynamic subdomains also

You noticed the comments 'subdomains_1' and so on, this is because almost all book's code shots, are running examples.

You can find them by pressing here.

Named Parameters

Named parameters are just custom paths to your routes, you can access them for each request using context's **c.Param("nameoftheparameter")**. Get all, as array (**{Key,Value}**) using **c.Params** property.

No limit on how long a path can be.

Usage:

```
package main

import (
    "strconv"

    "github.com/kataras/iris"
)

func main() {
    // Match to /hello/iris,  (if PathCorrection:true match also
 /hello/iris/)
    // Not match to /hello or /hello/ or /hello/iris/something
    iris.Get("/hello/:name", func(c *iris.Context) {
        // Retrieve the parameter name
        name := c.Param("name")
        c.Write("Hello %s", name)
    })

    // Match to /profile/iris/friends/1, (if PathCorrection:true
 match also /profile/iris/friends/1/)
    // Not match to /profile/ , /profile/iris ,
    // Not match to /profile/iris/friends,  /profile/iris/friend
s ,
    // Not match to /profile/iris/friends/2/something
    iris.Get("/profile/:fullname/friends/:friendID", func(c *iri
s.Context) {
        // Retrieve the parameters fullname and friendID
        fullname := c.Param("fullname")
```

```
        friendID, err := c.ParamInt("friendID")
        if err != nil {
            // Do something with the error
        }
        c.HTML(iris.StatusOK, "<b> Hello </b>"+fullname+"<b> wit
h friends ID </b>"+strconv.Itoa(friendID))
    })

    /* Example: /posts/:id and /posts/new (dynamic value confict
s with the static 'new') for performance reasons and simplicity
       but if you need to have them you can do that: */

    iris.Get("/posts/*action", func(ctx *iris.Context) {
        action := ctx.Param("action")
        if action == "/new" {
            // it's posts/new page
            ctx.Write("POSTS NEW")
        } else {
            ctx.Write("OTHER POSTS")
            // it's posts/:id page
            //doSomething with the action which is the id
        }
    })

    iris.Listen(":8080")
}
```

Match anything

```
// Will match any request which url's preffix is "/anything/" an
d has content after that
iris.Get("/anything/*randomName", func(c *iris.Context) { } )
// Match: /anything/whateverhere/whateveragain , /anything/blabl
abla
// c.Param("randomName") will be /whateverhere/whateveragain, bl
ablabla
// Not Match: /anything , /anything/ , /something
```

Static files

Serve a static directory

```
// StaticHandler returns a HandlerFunc to serve static system di
rectory
// Accepts 5 parameters
//
// first is the systemPath (string)
// Path to the root directory to serve files from.
//
// second is the  stripSlashes (int) level
// * stripSlashes = 0, original path: "/foo/bar", result: "/foo/
bar"
// * stripSlashes = 1, original path: "/foo/bar", result: "/bar"
// * stripSlashes = 2, original path: "/foo/bar", result: ""
//
// third is the compress (bool)
// Transparently compresses responses if set to true.
//
// The server tries minimizing CPU usage by caching compressed f
iles.
// It adds FSCompressedFileSuffix suffix to the original file na
me and
// tries saving the resulting compressed file under the new file
 name.
// So it is advisable to give the server write access to Root
// and to all inner folders in order to minimze CPU usage when s
erving
// compressed responses.
//
// fourth is the generateIndexPages (bool)
// Index pages for directories without files matching IndexNames
// are automatically generated if set.
//
// Directory index generation may be quite slow for directories
// with many files (more than 1K), so it is discouraged enabling
```

```
// index pages' generation for such directories.
//
// fifth is the indexNames ([]string)
// List of index file names to try opening during directory acce
ss.
//
// For example:
//
//     * index.html
//     * index.htm
//     * my-super-index.xml
//
StaticHandler(systemPath string, stripSlashes int, compress bool
,
                  generateIndexPages bool, indexNames []string)
HandlerFunc

// Static registers a route which serves a system directory
// this doesn't generates an index page which list all files
// no compression is used also, for these features look at Stati
cFS func
// accepts three parameters
// first parameter is the request url path (string)
// second parameter is the system directory (string)
// third parameter is the level (int) of stripSlashes
// * stripSlashes = 0, original path: "/foo/bar", result: "/foo/
bar"
// * stripSlashes = 1, original path: "/foo/bar", result: "/bar"
// * stripSlashes = 2, original path: "/foo/bar", result: ""
Static(relative string, systemPath string, stripSlashes int)

// StaticFS registers a route which serves a system directory
// generates an index page which list all files
// uses compression which file cache, if you use this method it
will generate compressed files also
// think this function as small fileserver with http
// accepts three parameters
// first parameter is the request url path (string)
// second parameter is the system directory (string)
// third parameter is the level (int) of stripSlashes
```

```
// * stripSlashes = 0, original path: "/foo/bar", result: "/foo/
bar"
// * stripSlashes = 1, original path: "/foo/bar", result: "/bar"
// * stripSlashes = 2, original path: "/foo/bar", result: ""
StaticFS(relative string, systemPath string, stripSlashes int)

// StaticWeb same as Static but if index.html e
// xists and request uri is '/' then display the index.html's co
ntents
// accepts three parameters
// first parameter is the request url path (string)
// second parameter is the system directory (string)
// third parameter is the level (int) of stripSlashes
// * stripSlashes = 0, original path: "/foo/bar", result: "/foo/
bar"
// * stripSlashes = 1, original path: "/foo/bar", result: "/bar"
// * stripSlashes = 2, original path: "/foo/bar", result: ""
StaticWeb(relative string, systemPath string, stripSlashes int)

// StaticServe serves a directory as web resource
// it's the simpliest form of the Static* functions
// Almost same usage as StaticWeb
// accepts only one required parameter which is the systemPath
// ( the same path will be used to register the GET&HEAD routes)
// if second parameter is empty, otherwise the requestPath is th
e second parameter
// it uses gzip compression (compression on each request, no fil
e cache)
StaticServe(systemPath string, requestPath ...string)
```

```
iris.Static("/public", "./static/assets/", 1)
//-> /public/assets/favicon.ico
```

```
iris.StaticFS("/ftp", "./myfiles/public", 1)
```

```
iris.StaticWeb("/","./my_static_html_website", 1)
```

```
StaticServe(systemPath string, requestPath ...string)
```

Manual static file serving

```
// ServeFile serves a view file, to send a file
// to the client you should use the SendFile(serverfilename,clie
ntfilename)
// receives two parameters
// filename/path (string)
// gzipCompression (bool)
//
// You can define your own "Content-Type" header also, after thi
s function call
ServeFile(filename string, gzipCompression bool) error
```

Serve static individual file

```
iris.Get("/txt", func(ctx *iris.Context) {
    ctx.ServeFile("./myfolder/staticfile.txt", false)
}
```

For example if you want manual serve static individual files dynamically you can do something like that:

```
package main

import (
    "strings"
    "github.com/kataras/iris"
    "github.com/kataras/iris/utils"
)

func main() {

    iris.Get("/*file", func(ctx *iris.Context) {
            requestpath := ctx.Param("file")

            path := strings.Replace(requestpath, "/", utils.Path
Seperator, -1)

            if !utils.DirectoryExists(path) {
                ctx.NotFound()
                return
            }

            ctx.ServeFile(path, false) // make this true to use
gzip compression
    }
}

iris.Listen(":8080")
```

The previous example is almost identical with

```
StaticServe(systemPath string, requestPath ...string)
```

```
func main() {
  iris.StaticServe("./mywebpage")
  // Serves all files inside this directory to the GET&HEAD rout
e: 0.0.0.0:8080/mywebpage
  // using gzip compression ( no file cache, for file cache with
 zipped files use the StaticFS)
  iris.Listen(":8080")
}
```

```
func main() {
  iris.StaticServe("./static/mywebpage","/webpage")
  // Serves all files inside filesystem path ./static/mywebpage
to the GET&HEAD route: 0.0.0.0:8080/webpage
  iris.Listen(":8080")
}
```

Favicon

Imagine that we have a folder named `static` which has subfolder `favicons` and this folder contains a favicon, for example `iris_favicon_32_32.ico` .

```
// ./main.go
package main

import "github.com/kataras/iris"

func main() {
    iris.Favicon("./static/favicons/iris_favicon_32_32.ico")

    iris.Get("/", func(ctx *iris.Context) {
        ctx.HTML(iris.StatusOK, "You should see the favicon now
at the side of your browser.")
    })

    iris.Listen(":8080")
}
```

Practical example here

Send files

Send a file, force-download to the client

```
// You can define your own "Content-Type" header also, after thi
s function call
// for example: ctx.Response.Header.Set("Content-Type","theconte
nt/type")
SendFile(filename string, destinationName string) error
```

```
package main

import "github.com/kataras/iris"

func main() {

    iris.Get("/servezip", func(c *iris.Context) {
        file := "./files/first.zip"
        err := c.SendFile(file, "saveAsName.zip")
        if err != nil {
            println("error: " + err.Error())
        }
    })

    iris.Listen(":8080")
}
```

You can also send bytes manually, which will be downloaded by the user:

```
package main

import "github.com/kataras/iris"

func main() {

    iris.Get("/servezip", func(c *iris.Context) {
        // read your file or anything
        var binary data[]
        ctx.Data(iris.StatusOK, data)
    })

    iris.Listen(":8080")
}
```

Send e-mails

This is a package.

Sending plain or rich content e-mails is an easy process with Iris.

Configuration

```
// Config keeps the configs for mail sender service
type Config struct {
    // Host is the server mail host, IP or address
    Host string
    // Port is the listening port
    Port int
    // Username is the auth username@domain.com for the sender
    Username string
    // Password is the auth password for the sender
    Password string
    // FromAlias is the from part, if empty this is the first pa
rt before @ from the Username field
    FromAlias string
    // UseCommand enable it if you want to send e-mail with the
mail command  instead of smtp
    //
    // Host,Port & Password will be ignored
    // ONLY FOR UNIX
    UseCommand bool
}
```

```
Send(subject string, body string, to ...string) error
```

Example

File: `./main.go`

```
package main
```

```
import (
    "github.com/iris-contrib/mail"
    "github.com/kataras/iris"
)

func main() {
    // change these to your settings

    cfg := mail.Config{
        Host:     "smtp.mailgun.org",
        Username: "postmaster@sandbox661c307650f04e909150b37c0f3
b2f09.mailgun.org",
        Password: "38304272b8ee5c176d5961dc155b2417",
        Port:     587,
    }
    // change these to your e-mail to check if that works

    // create the service
    mailService := mail.New(cfg)

    var to = []string{"kataras2006@hotmail.com", "social@ideopod
.com"}

    // standalone

    //iris.Must(mailService.Send("iris e-mail test subject", "</
h1>outside of context before server's listen!</h1>", to...))

    //inside handler
    iris.Get("/send", func(ctx *iris.Context) {
        content := `<h1>Hello From Iris web framework</h1> <br/>
<br/> <span style="color:blue"> This is the rich message body </
span>`

        err := mailService.Send("iris e-mail just t3st subject",
 content, to...)

        if err != nil {
            ctx.HTML(200, "<b> Problem while sending the e-mail:
 "+err.Error())
```

```
        } else {
            ctx.HTML(200, "<h1> SUCCESS </h1>")
        }
    })

    // send a body by template
    iris.Get("/send/template", func(ctx *iris.Context) {
        content := iris.TemplateString("body.html", iris.Map{
            "Message": " his is the rich message body sent by a
template!!",
            "Footer":  "The footer of this e-mail!",
        }, iris.RenderOptions{"charset" :"UTF-8"})
            // iris.RenderOptions are optional parameter,
            // "charset" defaults to UTF-8 but you can change it
 for a
            // particular mail receiver

        err := mailService.Send("iris e-mail just t3st subject",
 content, to...)

        if err != nil {
            ctx.HTML(200, "<b> Problem while sending the e-mail:
 "+err.Error())
        } else {
            ctx.HTML(200, "<h1> SUCCESS </h1>")
        }
    })
    iris.Listen(":8080")
}
```

File: `./templates/body.html`

```
<h1>Hello From Iris web framework</h1>
<br/><br/>
<span style="color:red"> {{.Message}}</span>
<hr/>

<b> {{.Footer}} </b>
```

Render

Think the 'Render' as an action which sends\/responses with a rich content to the client.

The render actions, are separated in two iris-theoretical 'categories'

- Response content using Response Engines, by 'Content-Type\/ or Key', you will understand what key is, later.
- Templates using Template Engines, by 'filename'.

Response Engines

Easy and fast way to render any type of data. **JSON, JSONP, XML, Text, Data, Markdown** .or any custom type.

- examples are located here

Template Engines

Iris gives you the freedom to render templates through 6+ built'n template engines, you can create your own and 'inject' that to the iris station, you can also use more than one template engines at the same time (when the file extension is different from the other).

- examples are located here

Install

Install one response engine and all will be installed.

```
$ go get -u github.com/iris-contrib/response/json
```

Iris' Station configuration

Remember, when 'station' we mean the default `iris.$CALL` or `api:= iris.New(); api.$CALL`

```
iris.Config.Gzip = true // compressed gzip contents to the clien
t, the same for Template Engines also, defaults to false
iris.Config.Charset = "UTF-8" // defaults to "UTF-8", the same f
or Template Engines also
```

They can be overriden for specific `Render` action:

```
func(ctx *iris.Context){
 ctx.Render("any/contentType", anyValue{}, iris.RenderOptions{"g
zip":false, "charset": "UTF-8"})
}
```

How to use

First of all don't be scary about the 'big' article here, a response engine works very simple and is easy to understand how.

Let's see what are the built'n response by content-type context's methods using the defaults only, unchanged, response engines.

```

package main
```

```
import (
    "encoding/xml"

    "github.com/kataras/iris"
)

type ExampleXml struct {
    XMLName xml.Name `xml:"example"`
    One     string   `xml:"one,attr"`
    Two     string   `xml:"two,attr"`
}

func main() {
    iris.Get("/data", func(ctx *iris.Context) {
        ctx.Data(iris.StatusOK, []byte("Some binary data here.")
)
    })

    iris.Get("/text", func(ctx *iris.Context) {
        ctx.Text(iris.StatusOK, "Plain text here")
    })

    iris.Get("/json", func(ctx *iris.Context) {
        ctx.JSON(iris.StatusOK, map[string]string{"hello": "json"
}) // or myjsonStruct{hello:"json}
    })

    iris.Get("/jsonp", func(ctx *iris.Context) {
        ctx.JSONP(iris.StatusOK, "callbackName", map[string]stri
ng{"hello": "jsonp"})
    })

    iris.Get("/xml", func(ctx *iris.Context) {
        ctx.XML(iris.StatusOK, ExampleXml{One: "hello", Two: "xm
l"}) // or iris.Map{"One":"hello"...}
    })

    iris.Get("/markdown", func(ctx *iris.Context) {
        ctx.Markdown(iris.StatusOK, "# Hello Dynamic Markdown Ir
is")
```

```
    })

    iris.Listen(":8080")
}
```

Bellow you will, propably, see how 'good' are my english (joke...), but at the end we're coders and some of us programmers too, so I hope you will be able to understand at least, the code snippets (a lot of them, you will be tired from this simplicity).

Text Response Engine

```
package main

import "github.com/kataras/iris"

func main() {
    iris.Config.Charset = "UTF-8" // this is the default, you do
n't have to set it manually

    myString := "this is just a simple string which you can alre
ady render with ctx.Write"

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Text(iris.StatusOK, myString)
    })

    iris.Get("/alternative_1", func(ctx *iris.Context) {
        ctx.Render("text/plain", myString)
    })

    iris.Get("/alternative_2", func(ctx *iris.Context) {
        ctx.RenderWithStatus(iris.StatusOK, "text/plain", myStri
ng)
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.Render("text/plain", myString, iris.RenderOptions{"c
harset": "UTF-8"}) // default & global charset is UTF-8
    })

    iris.Get("/alternative_4", func(ctx *iris.Context) {
        // logs if any error and sends http status '500 internal
 server error' to the client
        ctx.MustRender("text/plain", myString)
    })

    iris.Listen(":8080")
}
```

Custom response engine

You can create a custom response engine using a func or an interface which implements the `iris.ResponseEngine` which contains a simple function: `Response(val interface{}, options ...map[string]interface{}) ([]byte, error)`

A custom engine can be used to register a totally new content writer for a known ContentType or for a custom ContentType

You can imagine its useful, I will show you one right now.

Let's do a 'trick' here, which works for all other response engines, custom or not:

say for example, that you want a static'footer/suffix' on your content.

IF a response engine has the same key and the same content type then the contents are appended and the final result will be rendered to the client .

Let's do this with `text/plain` content type, because you can see its results easly, the first engine will use this "text/plain" as key, the second & third will use the same, as firsts, key, which is the ContentType also.

```
package main

import (
    "github.com/iris-contrib/response/text"
    "github.com/kataras/iris"
)

func main() {
    // here we are registering the default text/plain,  and afte
r we will register the 'appender' only
    // we have to register the default because we will
    // add more response engines with the same content,
    // iris will not register this by-default if
    // other response engine with the corresponding ContentType
already exists

    iris.UseResponse(text.New(), text.ContentType) // it's the k
```

```
ey which happens to be a valid content-type also, "text/plain" s
o this will be used as the ContentType header

    // register a response engine: iris.ResponseEngine
    iris.UseResponse(&CustomTextEngine{}, text.ContentType)

    // register a response engine with func
    iris.UseResponse(iris.ResponseEngineFunc(func(val interface{
}, options ...map[string]interface{}) ([]byte, error) {
        return []byte("\nThis is the static SECOND AND LAST suff
ix!"), nil
    }), text.ContentType)

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Text(iris.StatusOK, "Hello!") // or ctx.Render(text.
ContentType," Hello!")
    })

    iris.Listen(":8080")
}

// This is the way you create one with raw iris.ResponseEngine i
mplementation:

// CustomTextEngine the response engine which appends a simple s
tring on the default's text engine
type CustomTextEngine struct{}

// Implement the iris.ResponseEngine
func (e *CustomTextEngine) Response(val interface{}, options ...
map[string]interface{}) ([]byte, error) {
    // we don't need the val, because we want only to append, so
 what we should do?
    // just return the []byte we want to be appended after the f
irst registered text/plain engine

    return []byte("\nThis is the static FIRST suffix!"), nil
}
```

iris.ResponseString

ResponseString gives you the result of the response engine's work, it doesn't renders to the client but you can use this function to collect the end result and send it via e-mail to the user, or anything you can imagine.

```
package main

import "github.com/kataras/iris"

func main() {
    markdownContents := `## Hello Markdown from Iris

This is an example of Markdown with Iris



Features
--------

All features of Sundown are supported, including:

*   **Compatibility**. The Markdown v1.0.3 test suite passes wit
h
    the --tidy option.  Without --tidy, the differences are
    mostly in whitespace and entity escaping, where blackfriday
is
    more consistent and cleaner.

*   **Common extensions**, including table support, fenced code
    blocks, autolinks, strikethroughs, non-strict emphasis, etc.

*   **Safety**. Blackfriday is paranoid when parsing, making it
safe
    to feed untrusted user input without fear of bad things
    happening. The test suite stress tests this and there are no
    known inputs that make it crash.  If you find one, please le
t me
    know and send me the input that does it.

    NOTE: "safety" in this context means *runtime safety only*.
In order to
```

```
    protect yourself against JavaScript injection in untrusted c
ontent, see
    [this example](https://github.com/russross/blackfriday#sanit
ize-untrusted-content).

*   **Fast processing**. It is fast enough to render on-demand i
n
    most web applications without having to cache the output.

*   **Thread safety**. You can run multiple parsers in different
    goroutines without ill effect. There is no dependence on glo
bal
    shared state.

*   **Minimal dependencies**. Blackfriday only depends on standa
rd
    library packages in Go. The source code is pretty
    self-contained, so it is easy to add to any project, includi
ng
    Google App Engine projects.

*   **Standards compliant**. Output successfully validates using
 the
    W3C validation tool for HTML 4.01 and XHTML 1.0 Transitional
.

    [this is a link](https://github.com/kataras/iris) `

    iris.Get("/", func(ctx *iris.Context) {
        // let's see
        // convert markdown string to html and print it to the l
ogger
        // THIS WORKS WITH ALL RESPONSE ENGINES, but I am not do
ing the same example for all engines again :) (the same you can
do with templates using the iris.TemplateString)
        htmlContents := iris.ResponseString("text/markdown", mar
kdownContents, iris.RenderOptions{"charset": "8859-1"}) // defau
lt is the iris.Config.Charset, which is UTF-8

        ctx.Log(htmlContents)
```

```
        ctx.Write("The Raw HTML is:\n%s", htmlContents)
    })

    iris.Listen(":8080")
}
```

Now we can continue to the rest of the default & built'n response engines

JSON Response Engine

```
package main

import "github.com/kataras/iris"

type myjson struct {
    Name string `json:"name"`
}

func main() {

    iris.Get("/", func(ctx *iris.Context) {
        ctx.JSON(iris.StatusOK, iris.Map{"name": "iris"})
    })

    iris.Get("/alternative_1", func(ctx *iris.Context) {
        ctx.JSON(iris.StatusOK, myjson{Name: "iris"})
    })

    iris.Get("/alternative_2", func(ctx *iris.Context) {
        ctx.Render("application/json", myjson{Name: "iris"})
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.RenderWithStatus(iris.StatusOK, "application/json",
myjson{Name: "iris"})
    })

    iris.Get("/alternative_4", func(ctx *iris.Context) {
        ctx.Render("application/json", myjson{Name: "iris"}, iri
```

```
s.RenderOptions{"charset": "UTF-8"}) // UTF-8 is the default.
    })

    iris.Get("/alternative_5", func(ctx *iris.Context) {
        // logs if any error and sends http status '500 internal
 server error' to the client
        ctx.MustRender("application/json", myjson{Name: "iris"},
 iris.RenderOptions{"charset": "UTF-8"}) // UTF-8 is the default.

    })

    iris.Listen(":8080")
}
```

```
package main

import (
    "github.com/iris-contrib/response/json"
    "github.com/kataras/iris"
)

type myjson struct {
    Name string `json:"name"`
}

func main() {
    iris.Config.Charset = "UTF-8" // this is the default, which
you can change

    //first example
    // use the json's Config, we need the import of the json res
ponse engine in order to change its internal configs
    // this is one of the reasons you need to import a default e
ngine,(template engine or response engine)
    /*
        type Config struct {
            Indent        bool
            UnEscapeHTML  bool
```

```
            Prefix        []byte
            StreamingJSON bool
        }
    */
    iris.UseResponse(json.New(json.Config{
        Prefix: []byte("MYPREFIX"),
    }), json.ContentType) // you can use anything as the second parameter, the json.ContentType is the string "application/json", the context.JSON renders with this engine's key.

    jsonHandlerSimple := func(ctx *iris.Context) {
        ctx.JSON(iris.StatusOK, myjson{Name: "iris"})
    }

    jsonHandlerWithRender := func(ctx *iris.Context) {
        // you can also change the charset for a specific render action with RenderOptions
        ctx.Render("application/json", myjson{Name: "iris"}, iris.RenderOptions{"charset": "8859-1"})
    }

    //second example,
    // imagine that we need the context.JSON to be listening to our "application/json" response engine with a custom prefix (we did that before)
    // but we also want a different renderer, but again application/json content type, with Indent option setted to true:
    iris.UseResponse(json.New(json.Config{Indent: true}), "json2")("application/json")
    // yes the UseResponse returns a function which you can map the content type if it's not declared on the key
    json2Handler := func(ctx *iris.Context) {
        ctx.Render("json2", myjson{Name: "My iris"})
    }

    iris.Get("/", jsonHandlerSimple)

    iris.Get("/render", jsonHandlerWithRender)

    iris.Get("/json2", json2Handler)
```

```
    iris.Listen(":8080")
}
```

JSONP Response Engine

```
package main

import "github.com/kataras/iris"

type myjson struct {
    Name string `json:"name"`
}

func main() {

    iris.Get("/", func(ctx *iris.Context) {
        ctx.JSONP(iris.StatusOK, "callbackName", iris.Map{"name"
: "iris"})
    })

    iris.Get("/alternative_1", func(ctx *iris.Context) {
        ctx.JSONP(iris.StatusOK, "callbackName", myjson{Name: "i
ris"})
    })

    iris.Get("/alternative_2", func(ctx *iris.Context) {
        ctx.Render("application/javascript", myjson{Name: "iris"
}, iris.RenderOptions{"callback": "callbackName"})
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.RenderWithStatus(iris.StatusOK, "application/javascr
ipt", myjson{Name: "iris"}, iris.RenderOptions{"callback": "call
backName"})
    })

    iris.Get("/alternative_4", func(ctx *iris.Context) {
        // logs if any error and sends http status '500 internal
```

```
 server error' to the client
        ctx.MustRender("application/javascript", myjson{Name: "i
ris"}, iris.RenderOptions{"callback": "callbackName", "charset":
"UTF-8"}) // UTF-8 is the default.
    })

    iris.Listen(":8080")
}
```

```
package main

import (
    "github.com/iris-contrib/response/jsonp"
    "github.com/kataras/iris"
)

type myjson struct {
    Name string `json:"name"`
}

func main() {
    iris.Config.Charset = "UTF-8" // this is the default, which
you can change

    //first example
    // this is one of the reasons you need to import a default e
ngine,(template engine or response engine)
    /*
        type Config struct {
            Indent   bool
            Callback string // the callback can be override by t
he context's options or parameter on context.JSONP
        }
    */
    iris.UseResponse(jsonp.New(jsonp.Config{
        Indent: true,
    }), jsonp.ContentType)
```

```
    // you can use anything as the second parameter,
    // the jsonp.ContentType is the string "application/javascri
pt",
    // the context.JSONP renders with this engine's key.

    handlerSimple := func(ctx *iris.Context) {
        ctx.JSONP(iris.StatusOK, "callbackName", myjson{Name: "i
ris"})
    }

    handlerWithRender := func(ctx *iris.Context) {
        // you can also change the charset for a specific render
 action with RenderOptions
        ctx.Render("application/javascript", myjson{Name: "iris"
}, iris.RenderOptions{"callback": "callbackName", "charset": "88
59-1"})
    }

    //second example,
    // but we also want a different renderer, but again "applica
tion/javascript" as content type, with Callback option setted gl
obaly:
    iris.UseResponse(jsonp.New(jsonp.Config{Callback: "callbackN
ame"}), "jsonp2")("application/javascript")
    // yes the UseResponse returns a function which you can map
the content type if it's not declared on the key
    handlerJsonp2 := func(ctx *iris.Context) {
        ctx.Render("jsonp2", myjson{Name: "My iris"})
    }

    iris.Get("/", handlerSimple)

    iris.Get("/render", handlerWithRender)

    iris.Get("/jsonp2", handlerJsonp2)

    iris.Listen(":8080")
}
```

XML Response Engine

```
package main

import "github.com/kataras/iris"

type myxml struct {
    XMLName xml.Name `xml:"xml_example"`
    First   string   `xml:"first,attr"`
    Second  string   `xml:"second,attr"`
}

func main() {

    iris.Get("/", func(ctx *iris.Context) {
        ctx.XML(iris.StatusOK, iris.Map{"first": "first attr ",
"second": "second attr"})
    })

    iris.Get("/alternative_1", func(ctx *iris.Context) {
        ctx.XML(iris.StatusOK, myxml{First: "first attr", Second
: "second attr"})
    })

    iris.Get("/alternative_2", func(ctx *iris.Context) {
        ctx.Render("text/xml", myxml{First: "first attr", Second
: "second attr"})
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.RenderWithStatus(iris.StatusOK, "text/xml", myxml{Fi
rst: "first attr", Second: "second attr"})
    })

    iris.Get("/alternative_4", func(ctx *iris.Context) {
        ctx.Render("text/xml", myxml{First: "first attr", Second
: "second attr"}, iris.RenderOptions{"charset": "UTF-8"}) // UTF
-8 is the default.
    })

    iris.Get("/alternative_5", func(ctx *iris.Context) {
```

```
        // logs if any error and sends http status '500 internal
 server error' to the client
        ctx.MustRender("text/xml", myxml{First: "first attr", Se
cond: "second attr"}, iris.RenderOptions{"charset": "UTF-8"})
    })

    iris.Listen(":8080")
}
```

```
package main

import (
    encodingXML "encoding/xml"

    "github.com/iris-contrib/response/xml"
    "github.com/kataras/iris"
)

type myxml struct {
    XMLName encodingXML.Name `xml:"xml_example"`
    First   string           `xml:"first,attr"`
    Second  string           `xml:"second,attr"`
}

func main() {
    iris.Config.Charset = "UTF-8" // this is the default, which
you can change

    //first example
    // this is one of the reasons you need to import a default e
ngine,(template engine or response engine)
    /*
        type Config struct {
            Indent bool
            Prefix []byte
        }
    */
    iris.UseResponse(xml.New(xml.Config{
        Indent: true,
```

```
    }), xml.ContentType)
    // you can use anything as the second parameter,
    // the jsonp.ContentType is the string "text/xml",
    // the context.XML renders with this engine's key.

    handlerSimple := func(ctx *iris.Context) {
        ctx.XML(iris.StatusOK, myxml{First: "first attr", Second
: "second attr"})
    }

    handlerWithRender := func(ctx *iris.Context) {
        // you can also change the charset for a specific render
 action with RenderOptions
        ctx.Render("text/xml", myxml{First: "first attr", Second
: "second attr"}, iris.RenderOptions{"charset": "8859-1"})
    }

    //second example,
    // but we also want a different renderer, but again "text/xm
l" as content type, with prefix option setted by configuration:
    iris.UseResponse(xml.New(xml.Config{Prefix: []byte("")}), "x
ml2")("text/xml") // if you really use a PREFIX it will be not v
alid xml, use it only for special cases
    // yes the UseResponse returns a function which you can map
the content type if it's not declared on the key
    handlerXML2 := func(ctx *iris.Context) {
        ctx.Render("xml2", myxml{First: "first attr", Second: "s
econd attr"})
    }

    iris.Get("/", handlerSimple)

    iris.Get("/render", handlerWithRender)

    iris.Get("/xml2", handlerXML2)

    iris.Listen(":8080")
}
```

Markdown Response Engine

```
package main

import "github.com/kataras/iris"

type myjson struct {
    Name string `json:"name"`
}

func main() {
    markdownContents := `## Hello Markdown from Iris

This is an example of Markdown with Iris



Features
--------

All features of Sundown are supported, including:

*   **Compatibility**. The Markdown v1.0.3 test suite passes wit
h
    the --tidy option.  Without --tidy, the differences are
    mostly in whitespace and entity escaping, where blackfriday
is
    more consistent and cleaner.

*   **Common extensions**, including table support, fenced code
    blocks, autolinks, strikethroughs, non-strict emphasis, etc.

*   **Safety**. Blackfriday is paranoid when parsing, making it
safe
    to feed untrusted user input without fear of bad things
    happening. The test suite stress tests this and there are no
    known inputs that make it crash.  If you find one, please le
t me
    know and send me the input that does it.
```

```
    NOTE: "safety" in this context means *runtime safety only*.
In order to
    protect yourself against JavaScript injection in untrusted c
ontent, see
    [this example](https://github.com/russross/blackfriday#sanit
ize-untrusted-content).

*   **Fast processing**. It is fast enough to render on-demand i
n
    most web applications without having to cache the output.

*   **Thread safety**. You can run multiple parsers in different
    goroutines without ill effect. There is no dependence on glo
bal
    shared state.

*   **Minimal dependencies**. Blackfriday only depends on standa
rd
    library packages in Go. The source code is pretty
    self-contained, so it is easy to add to any project, includi
ng
    Google App Engine projects.

*   **Standards compliant**. Output successfully validates using
 the
    W3C validation tool for HTML 4.01 and XHTML 1.0 Transitional
.

    [this is a link](https://github.com/kataras/iris) `

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Markdown(iris.StatusOK, markdownContents)
    })

    iris.Get("/alternative_1", func(ctx *iris.Context) {
        htmlContents := ctx.MarkdownString(markdownContents)
        ctx.HTML(iris.StatusOK, htmlContents)
    })

    // text/markdown is just the key which the markdown response
```

```
 engine and ctx.Markdown communicate,
    // it's real content type is text/html
    iris.Get("/alternative_2", func(ctx *iris.Context) {
        ctx.Render("text/markdown", markdownContents)
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.RenderWithStatus(iris.StatusOK, "text/markdown", mar
kdownContents)
    })

    iris.Get("/alternative_4", func(ctx *iris.Context) {
        ctx.Render("text/markdown", markdownContents, iris.Rende
rOptions{"charset": "UTF-8"}) // UTF-8 is the default.
    })

    iris.Get("/alternative_5", func(ctx *iris.Context) {
        // logs if any error and sends http status '500 internal
 server error' to the client
        ctx.MustRender("text/markdown", markdownContents, iris.R
enderOptions{"charset": "UTF-8"}) // UTF-8 is the default.
    })

    iris.Listen(":8080")
}
```

```
package main

import (
    "github.com/iris-contrib/response/markdown"
    "github.com/kataras/iris"
)

func main() {
    markdownContents := `## Hello Markdown from Iris

This is an example of Markdown with Iris
```

```
Features
--------

All features of Sundown are supported, including:

*   **Compatibility**. The Markdown v1.0.3 test suite passes wit
h
    the --tidy option.  Without --tidy, the differences are
    mostly in whitespace and entity escaping, where blackfriday
is
    more consistent and cleaner.

*   **Common extensions**, including table support, fenced code
    blocks, autolinks, strikethroughs, non-strict emphasis, etc.

*   **Safety**. Blackfriday is paranoid when parsing, making it
safe
    to feed untrusted user input without fear of bad things
    happening. The test suite stress tests this and there are no
    known inputs that make it crash.  If you find one, please le
t me
    know and send me the input that does it.

    NOTE: "safety" in this context means *runtime safety only*.
In order to
    protect yourself against JavaScript injection in untrusted c
ontent, see
    [this example](https://github.com/russross/blackfriday#sanit
ize-untrusted-content).

*   **Fast processing**. It is fast enough to render on-demand i
n
    most web applications without having to cache the output.

*   **Thread safety**. You can run multiple parsers in different
    goroutines without ill effect. There is no dependence on glo
bal
    shared state.
```

```
*   **Minimal dependencies**. Blackfriday only depends on standa
rd
    library packages in Go. The source code is pretty
    self-contained, so it is easy to add to any project, includi
ng
    Google App Engine projects.

*   **Standards compliant**. Output successfully validates using
 the
    W3C validation tool for HTML 4.01 and XHTML 1.0 Transitional
.

    [this is a link](https://github.com/kataras/iris) `

    //first example
    // this is one of the reasons you need to import a default e
ngine,(template engine or response engine)
    /*
        type Config struct {
            MarkdownSanitize bool
        }
    */
    iris.UseResponse(jsonp.New(jsonp.Config{
        Indent: true,
    }), jsonp.ContentType)
    // you can use anything as the second parameter,
    // the jsonp.ContentType is the string "text/markdown",
    // the context.Markdown renders with this engine's key.

    handlerWithRender := func(ctx *iris.Context) {
        // you can also change the charset for a specific render
 action with RenderOptions
        ctx.Render("text/markdown", markdownContents, iris.Rende
rOptions{"charset": "8859-1"})
    }

    //second example,
    // but we also want a different renderer, but again "text/ma
rkdown" as 'content type' (this is converted to text/html behind
 the scenes), with MarkdownSanitize option setted to true:
```

```
    iris.UseResponse(markdown.New(markdown.Config{MarkdownSaniti
ze: true}), "markdown2")("text/markdown")
    // yes the UseResponse returns a function which you can map
the content type if it's not declared on the key
    handlerMarkdown2 := func(ctx *iris.Context) {
        ctx.Render("markdown2", markdownContents)
    }

    iris.Get("/", handlerWithRender)

    iris.Get("/markdown2", handlerMarkdown2)

    iris.Listen(":8080")
}
```

Data(Binary) Response Engine

```
package main

import "github.com/kataras/iris"

func main() {
    myData := []byte("some binary data or a program here which w
ill not be a simple string at the production")

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Data(iris.StatusOK, myData)
    })

    iris.Get("/alternative_1", func(ctx *iris.Context) {
        ctx.Render("application/octet-stream", myData)
    })

    iris.Get("/alternative_2", func(ctx *iris.Context) {
        ctx.RenderWithStatus(iris.StatusOK, "application/octet-s
tream", myData)
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.Render("application/octet-stream", myData, iris.Rend
erOptions{"gzip": true}) // gzip is false by default
    })

    iris.Get("/alternative_4", func(ctx *iris.Context) {
        // logs if any error and sends http status '500 internal
 server error' to the client
        ctx.MustRender("application/octet-stream", myData)
    })

    iris.Listen(":8080")
}
```

- examples are located here

- You can contribute to create more response engines for Iris, click here to navigate to the reository.

Install

Install one template engine and all will be installed.

```
$ go get -u github.com/iris-contrib/template/html
```

Iris' Station configuration

Remember, when 'station' we mean the default `iris.$CALL` or `api:= iris.New(); api.$CALL`

```
iris.Config.IsDevelopment = true // reloads the templates on eac
h request, defaults to false
iris.Config.Gzip  = true // compressed gzip contents to the clie
nt, the same for Response Engines also, defaults to false
iris.Config.Charset = "UTF-8" // defaults to "UTF-8", the same f
or Response Engines also
```

The last two options (Gzip, Charset) can be overriden for specific 'Render' action:

```
func(ctx *iris.Context){
    ctx.Render("templateFile.html", anyBindingStruct{}, iris.Ren
derOptions{"gzip":false, "charset": "UTF-8"})
}
```

How to use

Most examples are written for the HTML Template Engine(default and built'n template engine for iris) but works for the rest of the engines also.

You will see first the template file's code, after the main.go code

HTML Template Engine, and general

```
<!-- ./templates/hi.html -->

<html>
<head>
<title>Hi Iris [THE TITLE]</title>
</head>
<body>
    <h1>Hi {{.Name}}
</body>
</html>
```

```
// ./main.go
package main

import "github.com/kataras/iris"

// nothing to do, defaults to ./templates and .html extension, n
o need to import any template engine because HTML engine is the
default
// if anything else has been registered
func main() {
    iris.Config.IsDevelopment = true // this will reload the tem
plates on each request, defaults to false
    iris.Get("/hi", hi)
    iris.Listen(":8080")
}

func hi(ctx *iris.Context) {
    ctx.MustRender("hi.html", struct{ Name string }{Name: "iris"
})
}
```

```
<!-- ./templates/layout.html -->
<html>
<head>
<title>My Layout</title>

</head>
<body>
    <h1>Body is:</h1>
    <!-- Render the current template here -->
    {{ yield }}
</body>
</html>
```

```
 <!-- ./templates/mypage.html -->
<h1>
    Title: {{.Title}}
</h1>
<h3>Message : {{.Message}} </h3>
```

```
// ./main.go
package main

import (
    "github.com/iris-contrib/template/html"
    "github.com/kataras/iris"
)

type mypage struct {
    Title   string
    Message string
}

func main() {

    iris.UseTemplate(html.New(html.Config{
        Layout: "layout.html",
    })).Directory("./templates", ".html") // the .Directory() is
 optional also, defaults to ./templates, .html
    // Note for html: this is the default iris' templaet engine,
 if zero engines added, then the template/html will be used auto
matically
    // These lines are here to show you how you can change its d
efault configuration

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("mypage.html", mypage{"My Page title", "Hello
 world!"}, iris.RenderOptions{"gzip": true})
        // Note that: you can pass "layout" : "otherLayout.html"
 to bypass the config's Layout property or iris.NoLayout to disa
ble layout on this render action.
        // RenderOptions is an optional parameter
    })

    iris.Listen(":8080")
}
```

```
<!-- ./templates/layouts/layout.html -->
<html>
<head>
<title>Layout</title>

</head>
<body>
    <h1>This is the global layout</h1>
    <br />
    <!-- Render the current template here -->
    {{ yield }}
</body>
</html>
```

```
<!-- ./templates/layouts/mylayout.html -->
<html>
<head>
<title>my Layout</title>

</head>
<body>
    <h1>This is the layout for the /my/ and /my/other routes only
</h1>
    <br />
    <!-- Render the current template here -->
    {{ yield }}
</body>
</html>
```

```
<!-- ./templates/partials/page1_partial1.html -->
<div style="background-color: white; color: red">
    <h1>Page 1's Partial 1</h1>
</div>
```

```
<!-- ./templates/page1.html -->
<div style="background-color: black; color: blue">

    <h1>Page 1</h1>

    {{ render "partials/page1_partial1.html"}}

</div>
```

```
// ./main.go
package main

import (
    "github.com/iris-contrib/template/html"
    "github.com/kataras/iris"
)

func main() {
    // directory and extensions defaults to ./templates, .html f
or all template engines
    iris.UseTemplate(html.New(html.Config{Layout: "layouts/layou
t.html"}))
    //iris.Config.Render.Template.Gzip = true
    iris.Get("/", func(ctx *iris.Context) {
        if err := ctx.Render("page1.html", nil); err != nil {
            println(err.Error())
        }
    })

    // remove the layout for a specific route
    iris.Get("/nolayout", func(ctx *iris.Context) {
        if err := ctx.Render("page1.html", nil, iris.RenderOptio
ns{"layout": iris.NoLayout}); err != nil {
            println(err.Error())
        }
    })

    // set a layout for a party, .Layout should be BEFORE any Ge
```

```
t or other Handle party's method
    my := iris.Party("/my").Layout("layouts/mylayout.html")
    {
        my.Get("/", func(ctx *iris.Context) {
            ctx.MustRender("page1.html", nil)
        })
        my.Get("/other", func(ctx *iris.Context) {
            ctx.MustRender("page1.html", nil)
        })
    }

    iris.Listen(":8080")
}
```

```
<!-- ./templates/layouts/layout.html -->

<html>
<head>
<title>My Layout</title>

</head>
<body>
    <!-- Render the current template here -->
    {{ yield }}
</body>
</html>
```

```
<!-- ./templates/partials/page1_partial1.html -->
<div style="background-color: white; color: red">
    <h1>Page 1's Partial 1</h1>
</div>
```

```
<!-- ./templates/page1.html -->
<div style="background-color: black; color: blue">

    <h1>Page 1</h1>

    {{ render "partials/page1_partial1.html"}}

</div>
```

```
// ./main.go
package main

import (
    "github.com/iris-contrib/template/html"
    "github.com/kataras/iris"
)

func main() {
    // directory and extensions defaults to ./templates, .html f
or all template engines
    iris.UseTemplate(html.New(html.Config{Layout: "layouts/layou
t.html"}))

    iris.Get("/", func(ctx *iris.Context) {
        s := iris.TemplateString("page1.html", nil)
        ctx.Write("The plain content of the template is: %s", s)
    })

    iris.Listen(":8080")
}
```

```
<!-- ./templates/page.html -->
<a href="{{url "my-page1"}}">http://127.0.0.1:8080/mypath</a>
<br />
<br />
<a href="{{url "my-page2" "theParam1" "theParam2"}}">http://127.
0.0.1:8080/mypath2/:param1/:param2</a>
<br />
<br />
<a href="{{url "my-page3" "theParam1" "theParam2AfterStatic"}}">
http://127.0.0.1:8080/mypath3/:param1/statichere/:param2</a>
<br />
<br />
<a href="{{url "my-page4" "theParam1" "theparam2AfterStatic" "ot
herParam" "matchAnything"}}">http://127.0.0.1:8080/mypath4/:para
m1/statichere/:param2/:otherparam/*something</a>
<br />
<br />
<a href="{{url "my-page5" "theParam1" "theParam2AfterStatic" "ot
herParam" "matchAnythingAfterStatic"}}">http://127.0.0.1:8080/my
path5/:param1/statichere/:param2/:otherparam/anything/*anything</
a>
<br />
<br />
<a href="{{url "my-page6" .ParamsAsArray }}">http://127.0.0.1:80
80/mypath6/:param1/:param2/staticParam/:param3AfterStatic</a>
```

```
// ./main.go
// Package main an example on how to naming your routes & use th
e custom 'url' HTML Template Engine, same for other template eng
ines
// we don't need to import the iris-contrib/template/html becaus
e iris uses this as the default engine if no other template engi
ne has been registered.
package main

import (
    "github.com/kataras/iris"
)
```

```
func main() {

    iris.Get("/mypath", emptyHandler)("my-page1")
    iris.Get("/mypath2/:param1/:param2", emptyHandler)("my-page2"
)
    iris.Get("/mypath3/:param1/statichere/:param2", emptyHandler
)("my-page3")
    iris.Get("/mypath4/:param1/statichere/:param2/:otherparam/*s
omething", emptyHandler)("my-page4")

    // same with Handle/Func
    iris.HandleFunc("GET", "/mypath5/:param1/statichere/:param2/
:otherparam/anything/*anything", emptyHandler)("my-page5")

    iris.Get("/mypath6/:param1/:param2/staticParam/:param3AfterS
tatic", emptyHandler)("my-page6")

    iris.Get("/", func(ctx *iris.Context) {
        // for /mypath6...
        paramsAsArray := []string{"theParam1", "theParam2", "the
Param3"}

        if err := ctx.Render("page.html", iris.Map{"ParamsAsArra
y": paramsAsArray}); err != nil {
            panic(err)
        }
    })

    iris.Get("/redirect/:namedRoute", func(ctx *iris.Context) {
        routeName := ctx.Param("namedRoute")

        println("The full uri of " + routeName + "is: " + iris.U
RL(routeName))
        // if routeName == "my-page1"
        // prints: The full uri of my-page1 is: http://127.0.0.1
:8080/mypath
        ctx.RedirectTo(routeName)
        // http://127.0.0.1:8080/redirect/my-page1 will redirect
 to -> http://127.0.0.1:8080/mypath
```

```
    })

    iris.Listen(":8080")
}

func emptyHandler(ctx *iris.Context) {
    ctx.Write("Hello from %s.", ctx.PathString())

}
```

```
<!-- ./templates/page.html -->
<!-- the only difference between normal named routes and dynamic
 subdomains named routes is that the first argument of  url
is the subdomain part instead of named parameter-->

<a href="{{url "dynamic-subdomain1" "username1"}}">username1.127
.0.0.1:8080/mypath</a>
<br />
<br />
<a href="{{url "dynamic-subdomain2" "username2" "theParam1" "the
Param2"}}">username2.127.0.0.1:8080/mypath2/:param1/:param2</a>
<br />
<br />
<a href="{{url "dynamic-subdomain3" "username3" "theParam1" "the
Param2AfterStatic"}}">username3.127.0.0.1:8080/mypath3/:param1/s
tatichere/:param2</a>
<br />
<br />
<a href="{{url "dynamic-subdomain4" "username4" "theParam1" "the
param2AfterStatic" "otherParam" "matchAnything"}}">username4.127
.0.0.1:8080/mypath4/:param1/statichere/:param2/:otherparam/*some
thing</a>
<br />
<br />
<a href="{{url "dynamic-subdomain5" .ParamsAsArray }}" >username
5.127.0.0.1:8080/mypath6/:param1/:param2/staticParam/:param3Afte
rStatic</a>
```

I will add hosts files contens only once, here, you can imagine the rest.

File location is Windows: Drive:/Windows/system32/drivers/etc/hosts, on Linux: /etc/hosts

```
# localhost name resolution is handled within DNS itself.
127.0.0.1       localhost
::1             localhost
#-IRIS-For development machine, you have to configure your dns a
lso for online, search google how to do it if you don't know

127.0.0.1        username1.127.0.0.1
127.0.0.1        username2.127.0.0.1
127.0.0.1        username3.127.0.0.1
127.0.0.1        username4.127.0.0.1
127.0.0.1        username5.127.0.0.1
# note that you can always use custom subdomains
#-END IRIS-
```

```
// ./main.go
// Package main same example as template_html_4 but with wildcar
d subdomains
package main

import (
    "github.com/kataras/iris"
)

func main() {

    wildcard := iris.Party("*.")
    {
        wildcard.Get("/mypath", emptyHandler)("dynamic-subdomain
1")
        wildcard.Get("/mypath2/:param1/:param2", emptyHandler)("
dynamic-subdomain2")
        wildcard.Get("/mypath3/:param1/statichere/:param2", empt
yHandler)("dynamic-subdomain3")
        wildcard.Get("/mypath4/:param1/statichere/:param2/:other
```

```
param/*something", emptyHandler)("dynamic-subdomain4")
        wildcard.Get("/mypath5/:param1/:param2/staticParam/:para
m3AfterStatic", emptyHandler)("dynamic-subdomain5")
    }

    iris.Get("/", func(ctx *iris.Context) {
        // for dynamic_subdomain:8080/mypath5...
        // the first parameter is always the subdomain part
        paramsAsArray := []string{"username5", "theParam1", "the
Param2", "theParam3"}

        if err := ctx.Render("page.html", iris.Map{"ParamsAsArra
y": paramsAsArray}); err != nil {
            panic(err)
        }
    })

    iris.Get("/redirect/:namedRoute/:subdomain", func(ctx *iris.
Context) {
        routeName := ctx.Param("namedRoute")
        subdomain := ctx.Param("subdomain")
        println("The full uri of " + routeName + "is: " + iris.U
RL(routeName, subdomain))
        // if routeName == "dynamic-subdomain1" && subdomain ==
"username1"
        // prints: The full uri ofd ynamic-subdomain1 is: http:/
/username1.127.0.0.1:8080/mypath
        ctx.RedirectTo(routeName, subdomain) // the second param
eter is the arguments, the first argument for dynamic subdomains
 is the subdomain part, after this, the named parameters
        // http://127.0.0.1:8080/redirect/my-subdomain1 will red
irect to ->  http://username1.127.0.0.1:8080/mypath
    })

    iris.Listen("127.0.0.1:8080")
}

func emptyHandler(ctx *iris.Context) {
    ctx.Write("[SUBDOMAIN: %s]Hello from Path: %s.", ctx.Subdoma
in(), ctx.PathString())
```

```
}
```

Django Template Engine

```
<!-- ./templates/mypage.html -->
<html>
<head>
<title>Hello Django from Iris</title>

</head>
<body>
    {% if is_admin %}
    <p>{{username}} is an admin!</p>
    {% endif %}
</body>
</html>
```

```
// ./main.go
package main

import (
    "github.com/iris-contrib/template/django"
    "github.com/kataras/iris"
)

func main() {

    iris.UseTemplate(django.New()).Directory("./templates", ".ht
ml")

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("mypage.html", map[string]interface{}{"userna
me": "iris", "is_admin": true}, iris.RenderOptions{"gzip": true}
)
    })

    iris.Listen(":8080")
}
```

```
<!-- ./templates/page.html -->
<!-- the only difference between normal named routes and dynamic
 subdomains named routes is that the first argument of  url
is the subdomain part instead of named parameter-->
<a href="{{ url("dynamic-subdomain1","username1") }}">username1.
127.0.0.1:8080/mypath</a>
<br />
<br />
<a href="{{ url("dynamic-subdomain2","username2","theParam1","th
eParam2") }}">username2.127.0.0.1:8080/mypath2/:param1/:param2</a
>
<br />
<br />
<a href="{{ url("dynamic-subdomain3","username3","theParam1","th
eParam2AfterStatic") }}" >username3.127.0.0.1:8080/mypath3/:para
m1/statichere/:param2</a>
<br />
<br />
<a href="{{ url("dynamic-subdomain4","username4","theParam1","th
eparam2AfterStatic","otherParam","matchAnything") }}" >username4
.127.0.0.1:8080/mypath4/:param1/statichere/:param2/:otherparam/*
something</a>
<br />
<br />
```

```
// ./main.go
// Package main same example as template_html_5 but for django/p
ongo2
package main

import (
    "github.com/iris-contrib/template/django"
    "github.com/kataras/iris"
)

func main() {
    iris.UseTemplate(django.New())
```

```
    wildcard := iris.Party("*.")
    {
        wildcard.Get("/mypath", emptyHandler)("dynamic-subdomain
1")
        wildcard.Get("/mypath2/:param1/:param2", emptyHandler)("
dynamic-subdomain2")
        wildcard.Get("/mypath3/:param1/statichere/:param2", empt
yHandler)("dynamic-subdomain3")
        wildcard.Get("/mypath4/:param1/statichere/:param2/:other
param/*something", emptyHandler)("dynamic-subdomain4")
    }

    iris.Get("/", func(ctx *iris.Context) {
        // for dynamic_subdomain:8080/mypath5...
        // the first parameter is always the subdomain part

        if err := ctx.Render("page.html", nil); err != nil {
            panic(err)
        }
    })

    iris.Get("/redirect/:namedRoute/:subdomain", func(ctx *iris.
Context) {
        routeName := ctx.Param("namedRoute")
        subdomain := ctx.Param("subdomain")
        println("The full uri of " + routeName + "is: " + iris.U
RL(routeName, subdomain))
        // if routeName == "dynamic-subdomain1" && subdomain ==
"username1"
        // prints: The full uri ofd ynamic-subdomain1 is: http:/
/username1.127.0.0.1:8080/mypath
        ctx.RedirectTo(routeName, subdomain) // the second param
eter is the arguments, the first argument for dynamic subdomains
 is the subdomain part, after this, the named parameters
        // http://127.0.0.1:8080/redirect/my-subdomain1 will red
irect to ->  http://username1.127.0.0.1:8080/mypath
    })

    iris.Listen("127.0.0.1:8080")
}
```

```
func emptyHandler(ctx *iris.Context) {
    ctx.Write("[SUBDOMAIN: %s]Hello from Path: %s.", ctx.Subdoma
in(), ctx.PathString())
}
```

> Note that, you can see more django examples syntax by navigating here

Handlebars Template Engine

```
<!-- ./templates/layouts/layout.html -->

<html>
<head>
<title>Layout</title>

</head>
<body>
    <h1>This is the global layout</h1>
    <br />
    <!-- Render the current template here -->
    {{ yield }}
</body>
</html>
```

```
<!-- ./templates/layouts/mylayout.html -->
<html>
<head>
<title>my Layout</title>

</head>
<body>
    <h1>This is the layout for the /my/ and /my/other routes only
</h1>
    <br />
    <!-- Render the current template here -->
    {{ yield }}
</body>
</html>
```

```
<!-- ./templates/partials/home_partial.html -->
<div style="background-color: white; color: red">
    <h1>Home's' Partial here!!</h1>
</div>
```

```
<!-- ./templates/home.html -->
<div style="background-color: black; color: white">

    Name: {{boldme Name}} <br /> Type: {{boldme Type}} <br /> Pa
th:
    {{boldme Path}} <br />
    <hr />

    The partial is: {{ render "partials/home_partial.html"}}

</div>
```

```
// ./main.go
package main

import (
```

```
    "github.com/aymerick/raymond"
    "github.com/iris-contrib/template/handlebars"
    "github.com/kataras/iris"
)

type mypage struct {
    Title   string
    Message string
}

func main() {
    // set the configuration for this template engine  (all temp
late engines has its configuration)
    config := handlebars.DefaultConfig()
    config.Layout = "layouts/layout.html"
    config.Helpers["boldme"] = func(input string) raymond.SafeSt
ring {
        return raymond.SafeString("<b> " + input + "</b>")
    }

    // set the template engine
    iris.UseTemplate(handlebars.New(config)).Directory("./templa
tes", ".html") // or .hbs , whatever you want

    iris.Get("/", func(ctx *iris.Context) {
        // optionally, set a context  for the template
        ctx.Render("home.html", map[string]interface{}{"Name": "
Iris", "Type": "Web", "Path": "/"})

    })

    // remove the layout for a specific route using iris.NoLayout

    iris.Get("/nolayout", func(ctx *iris.Context) {
        if err := ctx.Render("home.html", nil, iris.RenderOption
s{"layout": iris.NoLayout}); err != nil {
            ctx.Write(err.Error())
        }
    })
```

```
    // set a layout for a party, .Layout should be BEFORE any Ge
t or other Handle party's method
    my := iris.Party("/my").Layout("layouts/mylayout.html")
    {
        my.Get("/", func(ctx *iris.Context) {
            // .MustRender -> same as .Render but logs the error
 if any and return status 500 on client
            ctx.MustRender("home.html", map[string]interface{}{"
Name": "Iris", "Type": "Web", "Path": "/my/"})
        })
        my.Get("/other", func(ctx *iris.Context) {
            ctx.MustRender("home.html", map[string]interface{}{"
Name": "Iris", "Type": "Web", "Path": "/my/other"})
        })
    }

    iris.Listen(":8080")
}

// Note than you can see more handlebars examples syntax by navi
gating to https://github.com/aymerick/raymond
```

> Note than you can see more handlebars examples syntax by navigating here

Pug/Jade Template Engine

```
<!-- ./templates/partials/page1_partial1.jade -->
#footer
  p Copyright (c) foobar
```

```
<!-- ./templates/page.jade -->
doctype html
html(lang=en)
    head
        meta(charset=utf-8)
        title Title
    body
        p ads
        ul
            li The name is {{bold .Name}}.
            li The age is {{.Age}}.

        range .Emails
            div An email is {{.}}

        with .Jobs
            range .
                div.
                 An employer is {{.Employer}}
                 and the role is {{.Role}}

        {{ render "partials/page1_partial1.jade"}}
```

```
// ./main.go
package main

import (
    "html/template"

    "github.com/iris-contrib/template/pug"
    "github.com/kataras/iris"
)

type Person struct {
    Name   string
    Age    int
    Emails []string
    Jobs   []*Job
```

```
}

type Job struct {
    Employer string
    Role     string
}

func main() {
    // set the configuration for this template engine  (all temp
late engines has its configuration)
    cfg := pug.DefaultConfig()
    cfg.Funcs["bold"] = func(content string) (template.HTML, err
or) {
        return template.HTML("<b>" + content + "</b>"), nil
    }

    iris.UseTemplate(pug.New(cfg)).
        Directory("./templates", ".jade")

    iris.Get("/", func(ctx *iris.Context) {

        job1 := Job{Employer: "Super Employer", Role: "Team lead
er"}
        job2 := Job{Employer: "Fast Employer", Role: "Project ma
nagment"}

        person := Person{
            Name:   "name1",
            Age:    50,
            Emails: []string{"email1@something.gr", "email2.anyt
hing@gmail.com"},
            Jobs:   []*Job{&job1, &job2},
        }
        ctx.MustRender("page.jade", person)

    })

    iris.Listen(":8080")
}
```

```
<!-- ./templates/page.jade -->
a(href='{{url "dynamic-subdomain1" "username1"}}') username1.127
.0.0.1:8080/mypath
p.
 a(href='{{url "dynamic-subdomain2" "username2" "theParam1" "the
Param2"}}') username2.127.0.0.1:8080/mypath2/:param1/:param2

p.
 a(href='{{url "dynamic-subdomain3" "username3" "theParam1" "the
Param2AfterStatic"}}') username3.127.0.0.1:8080/mypath3/:param1/
statichere/:param2

p.
 a(href='{{url "dynamic-subdomain4" "username4" "theParam1" "the
param2AfterStatic" "otherParam" "matchAnything"}}') username4.12
7.0.0.1:8080/mypath4/:param1/statichere/:param2/:otherparam/*som
ething

p.
 a(href='{{url "dynamic-subdomain5" .ParamsAsArray }}') username
5.127.0.0.1:8080/mypath6/:param1/:param2/staticParam/:param3Afte
rStatic
```

```
// ./main.go
// Package main same example as template_html_5 but for pug/jade
package main

import (
    "github.com/iris-contrib/template/pug"
    "github.com/kataras/iris"
)

func main() {
    iris.UseTemplate(pug.New()).Directory("./templates", ".jade"
)

    wildcard := iris.Party("*.")
    {
```

```
        wildcard.Get("/mypath", emptyHandler)("dynamic-subdomain
1")
        wildcard.Get("/mypath2/:param1/:param2", emptyHandler)("
dynamic-subdomain2")
        wildcard.Get("/mypath3/:param1/statichere/:param2", empt
yHandler)("dynamic-subdomain3")
        wildcard.Get("/mypath4/:param1/statichere/:param2/:other
param/*something", emptyHandler)("dynamic-subdomain4")
        wildcard.Get("/mypath5/:param1/:param2/staticParam/:para
m3AfterStatic", emptyHandler)("dynamic-subdomain5")
    }

    iris.Get("/", func(ctx *iris.Context) {
        // for dynamic_subdomain:8080/mypath5...
        // the first parameter is always the subdomain part
        paramsAsArray := []string{"username5", "theParam1", "the
Param2", "theParam3"}

        if err := ctx.Render("page.jade", iris.Map{"ParamsAsArra
y": paramsAsArray}); err != nil {
            panic(err)
        }
    })

    iris.Get("/redirect/:namedRoute/:subdomain", func(ctx *iris.
Context) {
        routeName := ctx.Param("namedRoute")
        subdomain := ctx.Param("subdomain")
        println("The full uri of " + routeName + "is: " + iris.U
RL(routeName, subdomain))
        // if routeName == "dynamic-subdomain1" && subdomain ==
"username1"
        // prints: The full uri ofd ynamic-subdomain1 is: http:/
/username1.127.0.0.1:8080/mypath
        ctx.RedirectTo(routeName, subdomain) // the second param
eter is the arguments, the first argument for dynamic subdomains
 is the subdomain part, after this, the named parameters
        // http://127.0.0.1:8080/redirect/my-subdomain1 will red
irect to ->  http://username1.127.0.0.1:8080/mypath
    })
```

```
    iris.Listen("127.0.0.1:8080")
}

func emptyHandler(ctx *iris.Context) {
    ctx.Write("[SUBDOMAIN: %s]Hello from Path: %s.", ctx.Subdoma
in(), ctx.PathString())
}

// Note than you can see more Pug/Jade syntax examples by naviga
ting to https://github.com/Joker/jade
```

> Note than you can see more Pug/Jade syntax examples by navigating here

```
<!-- ./templates/basic.amber -->
!!! 5
html
    head
        title Hello Amber from Iris

        meta[name="description"][value="This is a sample"]

        script[type="text/javascript"]
            var hw = "Hello #{Name}!"
            alert(hw)

        style[type="text/css"]
            body {
                background: maroon;
                color: white
            }

    body
        header#mainHeader
            ul
                li.active
                    a[href="/"] Main Page
                        [title="Main Page"]
            h1
                 | Hi #{Name}

        footer
            | Hey
            br
            | There
```

```
// ./main.go
package main

import (
    "github.com/iris-contrib/template/amber"
    "github.com/kataras/iris"
)

type mypage struct {
    Name string
}

func main() {

    iris.UseTemplate(amber.New()).Directory("./templates", ".amb
er")

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("basic.amber", mypage{"iris"}, iris.RenderOpt
ions{"gzip": true})
    })

    iris.Listen(":8080")
}
```

Custom template engine

Simply, you have to implement only **3 functions**, for load and execute the templates. One optionally (**Funcs() map[string]interface{}**) which is used to register the iris' helpers funcs like `{{ url }}` and `{{ urlpath }}` .

```
type (
    // TemplateEngine the interface that all template engines mu
st implement
    TemplateEngine interface {
        // LoadDirectory builds the templates, usually by direct
ory and extension but these are engine's decisions
        LoadDirectory(directory string, extension string) error
        // LoadAssets loads the templates by binary
        // assetFn is a func which returns bytes, use it to load
 the templates by binary
        // namesFn returns the template filenames
        LoadAssets(virtualDirectory string, virtualExtension str
ing, assetFn func(name string) ([]byte, error), namesFn func() []
string) error

        // ExecuteWriter finds, execute a template and write its
 result to the out writer
        // options are the optional runtime options can be passe
d by user
        // an example of this is the "layout" or "gzip" option
        ExecuteWriter(out io.Writer, name string, binding interf
ace{}, options ...map[string]interface{}) error
    }

    // TemplateEngineFuncs is optional interface for the Templat
eEngine
    // used to insert the Iris' standard funcs, see var 'usedFun
cs'
    TemplateEngineFuncs interface {
        // Funcs should returns the context or the funcs,
        // this property is used in order to register the iris'
helper funcs
        Funcs() map[string]interface{}
    }
)
```

The simplest implementation, which you can look as example, is the Markdown Engine, which is located here.

iris.TemplateString

Executes and parses the template but instead of rendering to the client, it returns the contents. Useful when you want to send a template via e-mail or anything you can imagine.

```
<!-- ./templates/mypage.html -->
<html>
<head>
<title>Hello Django from Iris</title>

</head>
<body>
    {% if is_admin %}
    <p>{{username}} is an admin!</p>
    {% endif %}
</body>
</html>
```

```
// ./main.go
package main

import (
    "github.com/iris-contrib/template/django"
    "github.com/kataras/iris"
)

func main() {

    iris.UseTemplate(django.New()).Directory("./templates", ".ht
ml")

    iris.Get("/", func(ctx *iris.Context) {
        // THIS WORKS WITH ALL TEMPLATE ENGINES, but I am not do
ing the same example for all engines again :) (the same you can
do with templates using the iris.ResponseString)
        rawHtmlContents := iris.TemplateString("mypage.html", map
[string]interface{}{"username": "iris", "is_admin": true}, iris.
RenderOptions{"charset": "UTF-8"}) // defaults to UTF-8 already
        ctx.Log(rawHtmlContents)
        ctx.Write("The Raw HTML is:\n%s", rawHtmlContents)
    })

    iris.Listen(":8080")
}
```

> Note that: iris.TemplateString can be called outside of the context also

---

- examples are located here
- You can contribute to create more template engines for Iris, click here to navigate to the reository.

Gzip

Gzip compression is easy.

For **auto-gzip** to all response and template engines, just set the `iris.Config.Gzip = true` , which you can also change for specific render options:

```
//...
context.Render("mytemplate.html", bindingStruct{}, iris.RenderOptions{"gzip": false})
context.Render("my-custom-response", iris.Map{"anything":"everything"} , iris.RenderOptions{"gzip": false})
```

```
// WriteGzip writes response with gzipped body to w.
//
// The method gzips response body and sets 'Content-Encoding: gz
ip'
// header before writing response to w.
//
// WriteGzip doesn't flush response to w for performance reasons.

WriteGzip(w *bufio.Writer) error


// WriteGzipLevel writes response with gzipped body to w.
//
// Level is the desired compression level:
//
//     * CompressNoCompression
//     * CompressBestSpeed
//     * CompressBestCompression
//     * CompressDefaultCompression
//
// The method gzips response body and sets 'Content-Encoding: gz
ip'
// header before writing response to w.
//
// WriteGzipLevel doesn't flush response to w for performance re
asons.
WriteGzipLevel(w *bufio.Writer, level int) error
```

How to use

```
iris.Get("/something", func(ctx *iris.Context){
    ctx.Response.WriteGzip(...)
})
```

Other

See Static files and learn how you can serve big files, assets or webpages with gzip compression.

Streaming

Do progressive rendering via multiple flushes, streaming.

```
// StreamWriter registers the given stream writer for populating
// response body.
//
//
// This function may be used in the following cases:
//
//     * if response body is too big (more than 10MB).
//     * if response body is streamed from slow external sources.

//     * if response body must be streamed to the client in chun
ks.
//     (aka `http server push`).
StreamWriter(cb func(writer *bufio.Writer))
```

Usage example

```
package main

import(
    "github.com/kataras/iris"
    "bufio"
    "time"
    "fmt"
)

func main() {
    iris.Any("/stream",func (ctx *iris.Context){
        ctx.StreamWriter(stream)
    })

    iris.Listen(":8080")
}

func stream(w *bufio.Writer) {
    for i := 0; i < 10; i++ {
            fmt.Fprintf(w, "this is a message number %d", i)

            // Do not forget flushing streamed data to the clien
t.
            if err := w.Flush(); err != nil {
                return
            }
            time.Sleep(time.Second)
        }
}
```

To achieve the oposite make use of the `StreamReader`

```
// StreamReader sets response body stream and, optionally body s
ize.
//
// If bodySize is >= 0, then the bodyStream must provide exactly
 bodySize bytes
// before returning io.EOF.
//
// If bodySize < 0, then bodyStream is read until io.EOF.
//
// bodyStream.Close() is called after finishing reading all body
 data
// if it implements io.Closer.
//
// See also StreamReader.
StreamReader(bodyStream io.Reader, bodySize int)
```

Cookies

Cookie management, even your little brother can do this!

```
// SetCookie adds a cookie
SetCookie(cookie *fasthttp.Cookie)

// SetCookieKV adds a cookie, receives just a key(string) and a
value(string)
SetCookieKV(key, value string)

// GetCookie returns cookie's value by it's name
// returns empty string if nothing was found
GetCookie(name string) string

// RemoveCookie removes a cookie by it's name/key
RemoveCookie(name string)


// VisitAllCookies takes a visitor which loops on each (request'
s) cookie key and value
//
// Note: the method ctx.Request.Header.VisitAllCookie by fastht
p, has a strange bug which I cannot solve at the moment.
// This is the reason which this function exists and should be u
sed instead of fasthttp's built'n.
VisitAllCookies(visitor func(key string, value string))
```

How to use

```
iris.Get("/set", func(c *iris.Context){
    c.SetCookieKV("name","iris")
    c.Write("Cookie has been setted.")
})

iris.Get("/get", func(c *iris.Context){
    name := c.GetCookie("name")
    c.Write("Cookie's value: %s", name)
})

iris.Get("/remove", func(c *iris.Context){
    if name := c.GetCookie("name"); name != "" {
       c.RemoveCookie("name")
    }
    c.Write("Cookie has been removed.")
})
```

Flash messages

A flash message is used in order to keep a message in session through one or several requests of the same user. By default, it is removed from session after it has been displayed to the user. Flash messages are usually used in combination with HTTP redirections, because in this case there is no view, so messages can only be displayed in the request that follows redirection.

A flash message has a name and a content (AKA key and value). It is an entry of a map. The name is a string: often "notice", "success", or "error", but it can be anything. The content is usually a string. You can put HTML tags in your message if you display it raw. You can also set the message value to a number or an array: it will be serialized and kept in session like a string.

```
// SetFlash sets a flash message, accepts 2 parameters the key(s
tring) and the value(string)
// the value will be available on the NEXT request
SetFlash(key string, value string)

// GetFlash get a flash message by it's key
// returns the value as string and an error
//
// if the cookie doesn't exists the string is empty and the erro
r is filled
// after the request's life the value is removed
GetFlash(key string) (value string, err error)

// GetFlashes returns all the flash messages for available for t
his request
GetFlashes() map[string]string
```

Example

```
package main

import (
    "github.com/kataras/iris"
)

func main() {

    iris.Get("/set", func(c *iris.Context) {
        c.SetFlash("name", "iris")
        c.Write("Message setted, is available for the next reque
st")
    })

    iris.Get("/get", func(c *iris.Context) {
        name, err := c.GetFlash("name")
        if err != nil {
            c.Write(err.Error())
            return
        }
        c.Write("Hello %s", name)
    })

    iris.Get("/test", func(c *iris.Context) {

        name, err := c.GetFlash("name")
        if err != nil {
            c.Write(err.Error())
            return
        }

        c.Write("Ok you are comming from /set ,the value of the
name is %s", name)
        c.Write(", and again from the same context: %s", name)

    })

    iris.Listen(":8080")
```

```
}
```

Body binder

Body binder reads values from the body and set them to a specific object.

```
// ReadJSON reads JSON from request's body
ReadJSON(jsonObject interface{}) error

// ReadXML reads XML from request's body
ReadXML(xmlObject interface{}) error

// ReadForm binds the formObject  to the requeste's form data
ReadForm(formObject interface{}) error
```

How to use

JSON

```
package main

import "github.com/kataras/iris"

type Company struct {
   Public     bool      `form:"public"`
   Website    url.URL   `form:"website"`
   Foundation time.Time `form:"foundation"`
   Name       string
   Location   struct {
     Country  string
     City     string
   }
   Products   []struct {
     Name string
     Type string
   }
   Founders   []string
   Employees  int64
}

func MyHandler(c *iris.Context) {
  if err := c.ReadJSON(&Company{}); err != nil {
      panic(err.Error())
  }
}

func main() {
  iris.Get("/bind_json", MyHandler)
  iris.Listen(":8080")
}
```

XML

```
package main

import "github.com/kataras/iris"

type Company struct {
   Public     bool
   Website    url.URL
   Foundation time.Time
   Name       string
   Location   struct {
     Country  string
     City     string
   }
   Products   []struct {
     Name string
     Type string
   }
   Founders   []string
   Employees  int64
}

func MyHandler(c *iris.Context) {
  if err := c.ReadXML(&Company{}); err != nil {
      panic(err.Error())
  }
}

func main() {
  iris.Get("/bind_xml", MyHandler)
  iris.Listen(":8080")
}
```

Form

Types

The supported field types in the destination struct are:

- `string`

- `bool`
- `int` , `int8` , `int16` , `int32` , `int64`
- `uint` , `uint8` , `uint16` , `uint32` , `uint64`
- `float32` , `float64`
- `slice` , `array`
- `struct` and `struct anonymous`
- `map`
- `interface{}`
- `time.Time`
- `url.URL`
- slices []string
- `custom types` to one of the above types
- a `pointer` to one of the above types

Custom Marshaling

Is possible unmarshaling data and the key of a map by the `encoding.TextUnmarshaler` interface.

Example

```
 //./main.go

package main

import (
    "fmt"

    "github.com/kataras/iris"
)

type Visitor struct {
    Username string
    Mail     string
    Data     []string `form:"mydata"`
}

func main() {

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("form.html", nil)
    })

    iris.Post("/form_action", func(ctx *iris.Context) {
        visitor := Visitor{}
        err := ctx.ReadForm(&visitor)
        if err != nil {
            fmt.Println("Error when reading form: " + err.Error(
))
        }
        fmt.Printf("\n Visitor: %v", visitor)
    })

    iris.Listen(":8080")
}
```

```
<!-- ./templates/form.html -->
<!DOCTYPE html>
<head>
<meta charset="utf-8">
</head>
<body>
<form action="/form_action" method="post">
<input type="text" name="Username" />
<br/>
<input type="text" name="Mail" /><br/>
<select multiple="multiple" name="mydata">
<option value='one'>One</option>
<option value='two'>Two</option>
<option value='three'>Three</option>
<option value='four'>Four</option>
</select>
<hr/>
<input type="submit" value="Send data" />

</form>
</body>
</html>
```

Example

In form html

- Use symbol `.` for access a field/key of a structure or map. (i.e, `struct.key`)
- Use `[int_here]` for access to index of a slice/array. (i.e, `struct.array[0]`)

```
<form method="POST">
  <input type="text" name="Name" value="Sony"/>
  <input type="text" name="Location.Country" value="Japan"/>
  <input type="text" name="Location.City" value="Tokyo"/>
  <input type="text" name="Products[0].Name" value="Playstation
4"/>
  <input type="text" name="Products[0].Type" value="Video games"
/>
  <input type="text" name="Products[1].Name" value="TV Bravia 32"
/>
  <input type="text" name="Products[1].Type" value="TVs"/>
  <input type="text" name="Founders[0]" value="Masaru Ibuka"/>
  <input type="text" name="Founders[0]" value="Akio Morita"/>
  <input type="text" name="Employees" value="90000"/>
  <input type="text" name="public" value="true"/>
  <input type="url" name="website" value="http://www.sony.net"/>
  <input type="date" name="foundation" value="1946-05-07"/>
  <input type="text" name="Interface.ID" value="12"/>
  <input type="text" name="Interface.Name" value="Go Programming
 Language"/>
  <input type="submit"/>
</form>
```

Backend

You can use the tag `form` if the name of a input of form starts lowercase.

```
package main

type InterfaceStruct struct {
    ID   int
    Name string
}

type Company struct {
  Public     bool      `form:"public"`
  Website    url.URL   `form:"website"`
  Foundation time.Time `form:"foundation"`
```

```
  Name       string
  Location   struct {
    Country  string
    City     string
  }
  Products   []struct {
    Name string
    Type string
  }
  Founders   []string
  Employees  int64

  Interface interface{}
}

func MyHandler(c *iris.Context) {
  m := Company{
      Interface: &InterfaceStruct{},
  }

  if err := c.ReadForm(&m); err != nil {
          panic(err.Error())
  }
}

func main() {
  iris.Get("/bind_form", MyHandler)
  iris.Listen(":8080")
}
```

Custom HTTP Errors

You can define your own handlers when http error occurs.

```
package main

import (
    "github.com/kataras/iris"
)

func main() {

    iris.OnError(iris.StatusInternalServerError, func(ctx *iris.
Context) {
        ctx.Write("CUSTOM 500 INTERNAL SERVER ERROR PAGE")
        iris.Logger.Printf("http status: 500 happened!")
    })

    iris.OnError(iris.StatusNotFound, func(ctx *iris.Context) {
        ctx.Write("CUSTOM 404 NOT FOUND ERROR PAGE")
        iris.Logger.Printf("http status: 404 happened!")
    })

    // emit the errors to test them
    iris.Get("/500", func(ctx *iris.Context) {
        ctx.EmitError(iris.StatusInternalServerError) // ctx.Pan
ic()
    })

    iris.Get("/404", func(ctx *iris.Context) {
        ctx.EmitError(iris.StatusNotFound) // ctx.NotFound()
    })

    println("Server is running at: 80")
    iris.Listen(":80")

}
```

Context

```
    IContext interface {
                // it contains all fasthttp's RequestCtx's functi
ons
                *fasthttp.RequestCtx
                // These are the iris' specific
        Param(string) string
        ParamInt(string) (int, error)
        ParamInt64(string) (int64, error)
        URLParam(string) string
        URLParamInt(string) (int, error)
        URLParamInt64(string) (int64, error)
        URLParams() map[string]string
        MethodString() string
        HostString() string
        Subdomain() string
        PathString() string
        RequestPath(bool) string
        RequestIP() string
        RemoteAddr() string
        RequestHeader(k string) string
        FormValueString(string) string
        FormValues(string) []string
        SetStatusCode(int)
        SetContentType(string)
        SetHeader(string, string)
        Redirect(string, ...int)
        RedirectTo(string, ...interface{})
        NotFound()
        Panic()
        EmitError(int)
        Write(string, ...interface{})
        HTML(int, string)
        Data(int, []byte) error
        RenderWithStatus(int, string, interface{}, ...map[string]
interface{}) error
```

```
        Render(string, interface{}, ...map[string]interface{}) e
rror
        MustRender(string, interface{}, ...map[string]interface{
})
        TemplateString(string, interface{}, ...map[string]interf
ace{}) string
        MarkdownString(string) string
        Markdown(int, string)
        JSON(int, interface{}) error
        JSONP(int, string, interface{}) error
        Text(int, string) error
        XML(int, interface{}) error
        ServeContent(io.ReadSeeker, string, time.Time, bool) err
or
        ServeFile(string, bool) error
        SendFile(string, string) error
        Stream(func(*bufio.Writer))
        StreamWriter(cb func(*bufio.Writer))
        StreamReader(io.Reader, int)
        ReadJSON(interface{}) error
        ReadXML(interface{}) error
        ReadForm(interface{}) error
        Get(string) interface{}
        GetString(string) string
        GetInt(string) int
        Set(string, interface{})
        VisitAllCookies(func(string, string))
        SetCookie(*fasthttp.Cookie)
        SetCookieKV(string, string)
        RemoveCookie(string)
        GetFlashes() map[string]string
        GetFlash(string) (string, error)
        SetFlash(string, string)
        Session() interface {
            ID() string
            Get(string) interface{}
            GetString(key string) string
            GetInt(key string) int
            GetAll() map[string]interface{}
            VisitAll(cb func(k string, v interface{}))
```

```
            Set(string, interface{})
            Delete(string)
            Clear()
        }
        SessionDestroy()
        Log(string, ...interface{})
        Reset(*fasthttp.RequestCtx)
        GetRequestCtx() *fasthttp.RequestCtx
        Clone() IContext
        Do()
        Next()
        StopExecution()
        IsStopped() bool
        GetHandlerName() string
    }
```

The examples will give you the direction.

Logger

This is a middleware

Logs the incoming requests

```
 New(theLogger *logger.Logger, config ...Config) iris.HandlerFun
c
```

How to use

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/middleware/logger"
)

/*
With configs:

errorLogger := logger.New(iris.Logger, logger.Config{
        EnableColors: false, //enable it to enable colors for al
l, disable colors by iris.Logger.ResetColors(), defaults to fals
e
        // Status displays status code
        Status: true,
        // IP displays request's remote address
        IP: true,
        // Method displays the http method
        Method: true,
        // Path displays the request path
        Path: true,
})

iris.Use(errorLogger)
```

```
With default configs:

iris.Use(logger.New(iris.Logger))
*/
func main() {

    iris.Use(logger.New(iris.Logger))

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Write("hello")
    })

    iris.Get("/1", func(ctx *iris.Context) {
        ctx.Write("hello")
    })

    iris.Get("/2", func(ctx *iris.Context) {
        ctx.Write("hello")
    })

    // log http errors
    errorLogger := logger.New(iris.Logger)


    iris.OnError(iris.StatusNotFound, func(ctx *iris.Context) {
        errorLogger.Serve(ctx)
        ctx.Write("My Custom 404 error page ")
    })
    //

    iris.Listen(":8080")

}
```

You can create your **own Logger** to use

```
import (
    "github.com/kataras/iris/logger"
     mLogger "github.com/iris-contrib/middleware/logger"
)

theLogger := logger.New(config.DefaultLogger())

iris.Use(mLogger.New(theLogger))
```

> Note that: The logger middleware uses the ColorBgOther and ColorFgOther fields.

The configuration struct for the `iris/logger` is the `iris/config/logger`

```
    Logger struct {
        // Out the (file) writer which the messages/logs will pr
inted to
        // Default is os.Stdout
        Out *os.File
        // Prefix the prefix for each message
        // Default is ""
        Prefix string
        // Disabled default is false
        Disabled bool

        // foreground colors single SGR Code

        // ColorFgDefault the foreground color for the normal me
ssage bodies
        ColorFgDefault int
        // ColorFgInfo the foreground  color for info messages
        ColorFgInfo int
        // ColorFgSuccess the foreground color for success messa
ges
        ColorFgSuccess int
        // ColorFgWarning the foreground color for warning messa
ges
        ColorFgWarning int
```

```
        // ColorFgDanger the foreground color for error messages
        ColorFgDanger int
        // OtherFgColor the foreground color for the rest of the
 message types
        ColorFgOther int

        // background colors single SGR Code

        // ColorBgDefault the background color for the normal me
ssages
        ColorBgDefault int
        // ColorBgInfo the background  color for info messages
        ColorBgInfo int
        // ColorBgSuccess the background color for success messa
ges
        ColorBgSuccess int
        // ColorBgWarning the background color for warning messa
ges
        ColorBgWarning int
        // ColorBgDanger the background color for error messages
        ColorBgDanger int
        // OtherFgColor the background color for the rest of the
 message types
        ColorBgOther int

        // banners are the force printed/written messages, doesn
't care about Disabled field

        // ColorFgBanner the foreground color for the banner
        ColorFgBanner int
    }
```

The `config.DefaultLogger()` returns `config.Logger` :

```
    return Logger{
        Out:      os.Stdout,
        Prefix:   "",
        Disabled: false,
        // foreground colors
        ColorFgDefault: int(color.FgHiWhite),
        ColorFgInfo:    int(color.FgHiCyan),
        ColorFgSuccess: int(color.FgHiGreen),
        ColorFgWarning: int(color.FgHiMagenta),
        ColorFgDanger:  int(color.FgHiRed),
        ColorFgOther:   int(color.FgHiYellow),
        // background colors
        ColorBgDefault: 0,
        ColorBgInfo:    0,
        ColorBgSuccess: 0,
        ColorBgWarning: 0,
        ColorBgDanger:  0,
        ColorBgOther:   0,
        // banner color
        ColorFgBanner: int(color.FgHiBlue),
    }
```

HTTP access control

This is a middleware.

Some security work for you between the requests.

Options

```
    // AllowedOrigins is a list of origins a cross-domain reques
t can be executed from.
    // If the special "*" value is present in the list, all orig
ins will be allowed.
    // An origin may contain a wildcard (*) to replace 0 or more
 characters
    // (i.e.: http://*.domain.com). Usage of wildcards implies a
 small performance penality.
    // Only one wildcard can be used per origin.
    // Default value is ["*"]
    AllowedOrigins []string
    // AllowOriginFunc is a custom function to validate the orig
in. It take the origin
    // as argument and returns true if allowed or false otherwis
e. If this option is
    // set, the content of AllowedOrigins is ignored.
    AllowOriginFunc func(origin string) bool
    // AllowedMethods is a list of methods the client is allowed
 to use with
    // cross-domain requests. Default value is simple methods (G
ET and POST)
    AllowedMethods []string
    // AllowedHeaders is list of non simple headers the client i
s allowed to use with
    // cross-domain requests.
    // If the special "*" value is present in the list, all head
ers will be allowed.
    // Default value is [] but "Origin" is always appended to th
e list.
    AllowedHeaders []string
```

```
    AllowedHeadersAll bool

    // ExposedHeaders indicates which headers are safe to expose
 to the API of a CORS
    // API specification
    ExposedHeaders []string
    // AllowCredentials indicates whether the request can includ
e user credentials like
    // cookies, HTTP authentication or client side SSL certifica
tes.
    AllowCredentials bool
    // MaxAge indicates how long (in seconds) the results of a p
reflight request
    // can be cached
    MaxAge int
    // OptionsPassthrough instructs preflight to let other poten
tial next handlers to
    // process the OPTIONS method. Turn this on if your applicat
ion handles OPTIONS.
    OptionsPassthrough bool
    // Debugging flag adds additional output to debug server sid
e CORS issues
    Debug bool
```

```
import "github.com/iris-contrib/middleware/cors"

cors.New(cors.Options{})
```

Example

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/middleware/cors"
)

func main() {

    crs := cors.New(cors.Options{}) // options here

    iris.Use(crs) // register the middleware

    iris.Get("/home", func(c *iris.Context) {
        // ...
    })

    iris.Listen(":8080")
}
```

Basic Authentication

This is a middleware.

HTTP Basic authentication (BA) implementation is the simplest technique for enforcing access controls to web resources because it doesn't require cookies, session identifiers, or login pages; rather, HTTP Basic authentication uses standard fields in the HTTP header, obviating the need for handshakes. Read more.

Simple example

```
package main

import (
    "github.com/iris-contrib/middleware/basicauth"
    "github.com/kataras/iris"
)

func main() {
    authentication := basicauth.Default(map[string]string{"myuse
rname": "mypassword", "mySecondusername": "mySecondpassword"})

    // to global iris.Use(authentication)
    // to party: iris.Party("/secret", authentication) { ... }

    // to routes
    iris.Get("/secret", authentication, func(ctx *iris.Context)
{
        username := ctx.GetString("user") // this can be changed
, you will see at the middleware_basic_auth_2 folder
        ctx.Write("Hello authenticated user: %s ", username)
    })

    iris.Get("/secret/profile", authentication, func(ctx *iris.C
ontext) {
        username := ctx.GetString("user")
        ctx.Write("Hello authenticated user: %s from localhost:8
080/secret/profile ", username)
    })

    iris.Get("/othersecret", authentication, func(ctx *iris.Cont
ext) {
        username := ctx.GetString("user")
        ctx.Write("Hello authenticated user: %s from localhost:8
080/othersecret ", username)
    })

    iris.Listen(":8080")
}
```

Configurable example

```
package main

import (
    "time"

    "github.com/iris-contrib/middleware/basicauth"
    "github.com/kataras/iris"
)

func main() {
    authConfig := basicauth.Config{
        Users:      map[string]string{"myusername": "mypassword"
, "mySecondusername": "mySecondpassword"},
        Realm:      "Authorization Required", // if you don't se
t it it's "Authorization Required"
        ContextKey: "mycustomkey",            // if you don't se
t it it's "user"
        Expires:    time.Duration(30) * time.Minute,
    }

    authentication := basicauth.New(authConfig)

    // to global iris.Use(authentication)
    // to routes
    /*
        iris.Get("/mysecret", authentication, func(ctx *iris.Con
text) {
            username := ctx.GetString("mycustomkey") //  the Con
textkey from the authConfig
            ctx.Write("Hello authenticated user: %s ", username)
        })
    */

    // to party

    needAuth := iris.Party("/secret", authentication)
    {
```

```
        needAuth.Get("/", func(ctx *iris.Context) {
            username := ctx.GetString("mycustomkey") //  the Con
textkey from the authConfig
            ctx.Write("Hello authenticated user: %s from localho
st:8080/secret ", username)
        })

        needAuth.Get("/profile", func(ctx *iris.Context) {
            username := ctx.GetString("mycustomkey") //  the Con
textkey from the authConfig
            ctx.Write("Hello authenticated user: %s from localho
st:8080/secret/profile ", username)
        })

        needAuth.Get("/settings", func(ctx *iris.Context) {
            username := ctx.GetString("mycustomkey") //  the Con
textkey from the authConfig
            ctx.Write("Hello authenticated user: %s from localho
st:8080/secret/settings ", username)
        })
    }

    iris.Listen(":8080")
}
```

OAuth, OAuth2

This is a plugin.

This plugin helps you to be able to connect your clients using famous websites login APIs, it is a bridge to the goth.

Supported Providers

```
Amazon
Bitbucket
Box
Cloud Foundry
Digital Ocean
Dropbox
Facebook
GitHub
Gitlab
Google+
Heroku
InfluxCloud
Instagram
Lastfm
Linkedin
OneDrive
Paypal
SalesForce
Slack
Soundcloud
Spotify
Steam
Stripe
Twitch
Twitter
Uber
Wepay
Yahoo
Yammer
```

How to use - high level

```
    configs := oauth.Config{
      Path: "/auth", //defaults to /auth

      GithubKey:    "YOUR_GITHUB_KEY",
      GithubSecret: "YOUR_GITHUB_SECRET",
      GithubName:   "github", // defaults to github

      FacebookKey:    "YOUR_FACEBOOK_KEY",
      FacebookSecret: "YOUR_FACEBOOK_KEY",
      FacebookName:   "facebook", // defaults to facebook
      //and so on... enable as many as you want
    }

    // create the plugin with our configs
    authentication := oauth.New(configs)
    // register the plugin to iris
    iris.Plugins.Add(authentication)

    // came from yourhost:port/configs.Path/theprovidername
    // this is the handler inside yourhost:port/configs.Path/the
providername/callback
    // you can do redirect to the authenticated url or whatever
you want to do
    authentication.Success(func(ctx *iris.Context) {
        user := authentication.User(ctx) // returns the goth.User

    })
    authentication.Fail(func(ctx *iris.Context){})
```

Example:

```
// main.go
package main

import (
    "sort"
    "strings"
```

```
    "github.com/iris-contrib/plugin/oauth"
    "github.com/kataras/iris"
)

// register your auth via configs, providers with non-empty valu
es will be registered to goth automatically by Iris
var configs = oauth.Config{
    Path: "/auth", //defaults to /oauth

    GithubKey:    "YOUR_GITHUB_KEY",
    GithubSecret: "YOUR_GITHUB_SECRET",
    GithubName:   "github", // defaults to github

    FacebookKey:    "YOUR_FACEBOOK_KEY",
    FacebookSecret: "YOUR_FACEBOOK_KEY",
    FacebookName:   "facebook", // defaults to facebook
}

func init() {
    iris.Config.Sessions.Provider = "memory"
}

// ProviderIndex ...
type ProviderIndex struct {
    Providers    []string
    ProvidersMap map[string]string
}

func main() {
    // create the plugin with our configs
    authentication := oauth.New(configs)
    // register the plugin to iris
    iris.Plugins.Add(authentication)

    m := make(map[string]string)
    m[configs.GithubName] = "Github" // same as authentication.C
onfig.GithubName
    m[configs.FacebookName] = "Facebook"

    var keys []string
```

```
    for k := range m {
        keys = append(keys, k)
    }
    sort.Strings(keys)

    providerIndex := &ProviderIndex{Providers: keys, ProvidersMa
p: m}

    // set a  login success handler( you can use more than one h
andler)
    // if user succeed to logged in
    // client comes here from: localhost:3000/config.RouteName/l
owercase_provider_name/callback 's first handler, but the  previ
ous url is the localhost:3000/config.RouteName/lowercase_provide
r_name
    authentication.Success(func(ctx *iris.Context) {
        // if user couldn't validate then server sends StatusUna
uthorized, which you can handle by:  authentication.Fail OR iris
.OnError(iris.StatusUnauthorized, func(ctx *iris.Context){})
        user := authentication.User(ctx)

        // you can get the url by the named-route 'oauth' which
you can change by Config's field: RouteName
        println("came from " + authentication.URL(strings.ToLowe
r(user.Provider)))
        ctx.Render("user.html", user)
    })

    // customize the error page using: authentication.Fail(func(
ctx *iris.Context){....})

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("index.html", providerIndex)
    })

    iris.Listen(":3000")
}
```

View:

```
<!-- ./templates/index.html -->

{{range $key,$value:=.Providers}}
    <p><a href="{{ url "oauth" $value}}">Log in with {{index $.P
roviderssMap $value}}</a></p>
{{end}}
```

```
<!-- ./templates/user.html -->
<p>Name: {{.Name}}</p>
<p>Email: {{.Email}}</p>
<p>NickName: {{.NickName}}</p>
<p>Location: {{.Location}}</p>
<p>AvatarURL: {{.AvatarURL}} <img src="{{.AvatarURL}}"></p>
<p>Description: {{.Description}}</p>
<p>UserID: {{.UserID}}</p>
<p>AccessToken: {{.AccessToken}}</p>
<p>ExpiresAt: {{.ExpiresAt}}</p>
<p>RefreshToken: {{.RefreshToken}}</p>
```

How to use - low level

Low-level is just the iris-contrib/gothic which is like the original goth but converted to work with Iris.

Example:

```
package main

import (
    "html/template"
    "os"

    "sort"

    "github.com/iris-contrib/gothic"
    "github.com/kataras/iris"
    "github.com/markbates/goth"
```

```
    "github.com/markbates/goth/providers/amazon"
    "github.com/markbates/goth/providers/bitbucket"
    "github.com/markbates/goth/providers/box"
    "github.com/markbates/goth/providers/digitalocean"
    "github.com/markbates/goth/providers/dropbox"
    "github.com/markbates/goth/providers/facebook"
    "github.com/markbates/goth/providers/github"
    "github.com/markbates/goth/providers/gitlab"
    "github.com/markbates/goth/providers/gplus"
    "github.com/markbates/goth/providers/heroku"
    "github.com/markbates/goth/providers/instagram"
    "github.com/markbates/goth/providers/lastfm"
    "github.com/markbates/goth/providers/linkedin"
    "github.com/markbates/goth/providers/onedrive"
    "github.com/markbates/goth/providers/paypal"
    "github.com/markbates/goth/providers/salesforce"
    "github.com/markbates/goth/providers/slack"
    "github.com/markbates/goth/providers/soundcloud"
    "github.com/markbates/goth/providers/spotify"
    "github.com/markbates/goth/providers/steam"
    "github.com/markbates/goth/providers/stripe"
    "github.com/markbates/goth/providers/twitch"
    "github.com/markbates/goth/providers/twitter"
    "github.com/markbates/goth/providers/uber"
    "github.com/markbates/goth/providers/wepay"
    "github.com/markbates/goth/providers/yahoo"
    "github.com/markbates/goth/providers/yammer"
)

func init() {
    iris.Config.Sessions.Provider = "memory" // or "redis" and c
onfigure the Redis Provider
}

func main() {
    goth.UseProviders(
        twitter.New(os.Getenv("TWITTER_KEY"), os.Getenv("TWITTER
_SECRET"), "http://localhost:3000/auth/twitter/callback"),
        // If you'd like to use authenticate instead of authoriz
e in Twitter provider, use this instead.
```

```
        // twitter.NewAuthenticate(os.Getenv("TWITTER_KEY"), os.
Getenv("TWITTER_SECRET"), "http://localhost:3000/auth/twitter/ca
llback"),

        facebook.New(os.Getenv("FACEBOOK_KEY"), os.Getenv("FACEB
OOK_SECRET"), "http://localhost:3000/auth/facebook/callback"),
        gplus.New(os.Getenv("GPLUS_KEY"), os.Getenv("GPLUS_SECRE
T"), "http://localhost:3000/auth/gplus/callback"),
        github.New(os.Getenv("GITHUB_KEY"), os.Getenv("GITHUB_SE
CRET"), "http://localhost:3000/auth/github/callback"),
        spotify.New(os.Getenv("SPOTIFY_KEY"), os.Getenv("SPOTIFY
_SECRET"), "http://localhost:3000/auth/spotify/callback"),
        linkedin.New(os.Getenv("LINKEDIN_KEY"), os.Getenv("LINKE
DIN_SECRET"), "http://localhost:3000/auth/linkedin/callback"),
        lastfm.New(os.Getenv("LASTFM_KEY"), os.Getenv("LASTFM_SE
CRET"), "http://localhost:3000/auth/lastfm/callback"),
        twitch.New(os.Getenv("TWITCH_KEY"), os.Getenv("TWITCH_SE
CRET"), "http://localhost:3000/auth/twitch/callback"),
        dropbox.New(os.Getenv("DROPBOX_KEY"), os.Getenv("DROPBOX
_SECRET"), "http://localhost:3000/auth/dropbox/callback"),
        digitalocean.New(os.Getenv("DIGITALOCEAN_KEY"), os.Geten
v("DIGITALOCEAN_SECRET"), "http://localhost:3000/auth/digitaloce
an/callback", "read"),
        bitbucket.New(os.Getenv("BITBUCKET_KEY"), os.Getenv("BIT
BUCKET_SECRET"), "http://localhost:3000/auth/bitbucket/callback"
),
        instagram.New(os.Getenv("INSTAGRAM_KEY"), os.Getenv("INS
TAGRAM_SECRET"), "http://localhost:3000/auth/instagram/callback"
),
        box.New(os.Getenv("BOX_KEY"), os.Getenv("BOX_SECRET"), "
http://localhost:3000/auth/box/callback"),
        salesforce.New(os.Getenv("SALESFORCE_KEY"), os.Getenv("S
ALESFORCE_SECRET"), "http://localhost:3000/auth/salesforce/callb
ack"),
        amazon.New(os.Getenv("AMAZON_KEY"), os.Getenv("AMAZON_SE
CRET"), "http://localhost:3000/auth/amazon/callback"),
        yammer.New(os.Getenv("YAMMER_KEY"), os.Getenv("YAMMER_SE
CRET"), "http://localhost:3000/auth/yammer/callback"),
        onedrive.New(os.Getenv("ONEDRIVE_KEY"), os.Getenv("ONEDR
IVE_SECRET"), "http://localhost:3000/auth/onedrive/callback"),
```

```
        //Pointed localhost.com to http://localhost:3000/auth/ya
hoo/callback through proxy as yahoo
        // does not allow to put custom ports in redirection uri
        yahoo.New(os.Getenv("YAHOO_KEY"), os.Getenv("YAHOO_SECRE
T"), "http://localhost.com"),
        slack.New(os.Getenv("SLACK_KEY"), os.Getenv("SLACK_SECRE
T"), "http://localhost:3000/auth/slack/callback"),
        stripe.New(os.Getenv("STRIPE_KEY"), os.Getenv("STRIPE_SE
CRET"), "http://localhost:3000/auth/stripe/callback"),
        wepay.New(os.Getenv("WEPAY_KEY"), os.Getenv("WEPAY_SECRE
T"), "http://localhost:3000/auth/wepay/callback", "view_user"),
        //By default paypal production auth urls will be used, p
lease set PAYPAL_ENV=sandbox as environment variable for testing
        //in sandbox environment
        paypal.New(os.Getenv("PAYPAL_KEY"), os.Getenv("PAYPAL_SE
CRET"), "http://localhost:3000/auth/paypal/callback"),
        steam.New(os.Getenv("STEAM_KEY"), "http://localhost:3000
/auth/steam/callback"),
        heroku.New(os.Getenv("HEROKU_KEY"), os.Getenv("HEROKU_SE
CRET"), "http://localhost:3000/auth/heroku/callback"),
        uber.New(os.Getenv("UBER_KEY"), os.Getenv("UBER_SECRET")
, "http://localhost:3000/auth/uber/callback"),
        soundcloud.New(os.Getenv("SOUNDCLOUD_KEY"), os.Getenv("S
OUNDCLOUD_SECRET"), "http://localhost:3000/auth/soundcloud/callb
ack"),
        gitlab.New(os.Getenv("GITLAB_KEY"), os.Getenv("GITLAB_SE
CRET"), "http://localhost:3000/auth/gitlab/callback"),
    )

    m := make(map[string]string)
    m["amazon"] = "Amazon"
    m["bitbucket"] = "Bitbucket"
    m["box"] = "Box"
    m["digitalocean"] = "Digital Ocean"
    m["dropbox"] = "Dropbox"
    m["facebook"] = "Facebook"
    m["github"] = "Github"
    m["gitlab"] = "Gitlab"
    m["soundcloud"] = "SoundCloud"
```

```
    m["spotify"] = "Spotify"
    m["steam"] = "Steam"
    m["stripe"] = "Stripe"
    m["twitch"] = "Twitch"
    m["uber"] = "Uber"
    m["wepay"] = "Wepay"
    m["yahoo"] = "Yahoo"
    m["yammer"] = "Yammer"
    m["gplus"] = "Google Plus"
    m["heroku"] = "Heroku"
    m["instagram"] = "Instagram"
    m["lastfm"] = "Last FM"
    m["linkedin"] = "Linkedin"
    m["onedrive"] = "Onedrive"
    m["paypal"] = "Paypal"
    m["twitter"] = "Twitter"
    m["salesforce"] = "Salesforce"
    m["slack"] = "Slack"

    var keys []string
    for k := range m {
        keys = append(keys, k)
    }
    sort.Strings(keys)

    providerIndex := &ProviderIndex{Providers: keys, ProvidersMa
p: m}

    iris.Get("/auth/:provider/callback", func(ctx *iris.Context)
 {

        user, err := gothic.CompleteUserAuth(ctx)
        if err != nil {
            ctx.SetStatusCode(iris.StatusUnauthorized)
            ctx.Write(err.Error())
            return
        }

        t, _ := template.New("foo").Parse(userTemplate)
        ctx.ExecuteTemplate(t, user)
```

```
    })

    iris.Get("/auth/:provider", func(ctx *iris.Context) {
        err := gothic.BeginAuthHandler(ctx)
        if err != nil {
            ctx.Log(err.Error())
        }
    })

    iris.Get("/", func(ctx *iris.Context) {
        t, _ := template.New("foo").Parse(indexTemplate)
        ctx.ExecuteTemplate(t, providerIndex)
    })
    iris.Listen(":3000")
}

// ProviderIndex ...
type ProviderIndex struct {
    Providers    []string
    ProvidersMap map[string]string
}

var indexTemplate = `{{range $key,$value:=.Providers}}
    <p><a href="/auth/{{$value}}">Log in with {{index $.Provider
sMap $value}}</a></p>
{{end}}`

var userTemplate = `
<p>Name: {{.Name}}</p>
<p>Email: {{.Email}}</p>
<p>NickName: {{.NickName}}</p>
<p>Location: {{.Location}}</p>
<p>AvatarURL: {{.AvatarURL}} <img src="{{.AvatarURL}}"></p>
<p>Description: {{.Description}}</p>
<p>UserID: {{.UserID}}</p>
<p>AccessToken: {{.AccessToken}}</p>
<p>ExpiresAt: {{.ExpiresAt}}</p>
<p>RefreshToken: {{.RefreshToken}}</p>
`
```

high level and low level, no performance differences

JSON Web Tokens

This is a middleware.

What is it?

JWT.io has a great introduction to JSON Web Tokens.

In short, it's a signed JSON object that does something useful (for example, authentication). It's commonly used for Bearer tokens in Oauth 2. A token is made of three parts, separated by .'s. The first two parts are JSON objects, that have been base64url encoded. The last part is the signature, encoded the same way.

The first part is called the header. It contains the necessary information for verifying the last part, the signature. For example, which encryption method was used for signing and what key was used.

The part in the middle is the interesting bit. It's called the Claims and contains the actual stuff you care about. Refer to the RFC for information about reserved keys and the proper way to add your own.

Example

```
package main

import (
    "github.com/dgrijalva/jwt-go"
    jwtmiddleware "github.com/iris-contrib/middleware/jwt"
    "github.com/kataras/iris"
)

func main() {

    myJwtMiddleware := jwtmiddleware.New(jwtmiddleware.Config{
        ValidationKeyGetter: func(token *jwt.Token) (interface{}
, error) {
```

```
            return []byte("My Secret"), nil
        },
        SigningMethod: jwt.SigningMethodHS256,
    })

    iris.Get("/ping", PingHandler)

    iris.Get("/secured/ping", myJwtMiddleware.Serve, SecuredPing
Handler)
    iris.Listen(":8080")

}

type Response struct {
    Text string `json:"text"`
}

func PingHandler(ctx *iris.Context) {
    response := Response{"All good. You don't need to be authent
icated to call this"}
    ctx.JSON(iris.StatusOK, response)
}

func SecuredPingHandler(ctx *iris.Context) {
    response := Response{"All good. You only get this message if
 you're authenticated"}
    // get the *jwt.Token which contains user information using:
    // user:= myJwtMiddleware.Get(ctx) or context.Get("jwt").(*j
wt.Token)
    ctx.JSON(iris.StatusOK, response)
}
```

Secure

This is a middleware

Secure is an HTTP middleware for Go that facilitates some quick security wins.

```
import "github.com/iris-contrib/middleware/secure"

secure.New(secure.Options{}) // options here
```

Example

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/middleware/secure"
)

func main() {
    s := secure.New(secure.Options{
        AllowedHosts:            []string{"ssl.example.com"},


        // AllowedHosts is a list of fully qualified domain names

        //that are allowed. Default is empty list,
        //which allows any and all host names.
        SSLRedirect:             true,

        // If SSLRedirect is set to true, then only allow HTTPS
requests.
        //Default is false.
        SSLTemporaryRedirect:    false,

        // If SSLTemporaryRedirect is true,
        //the a 302 will be used while redirecting.
```

```
        //Default is false (301).
        SSLHost:                "ssl.example.com",

        // SSLHost is the host name that is used to
        //redirect HTTP requests to HTTPS.
        //Default is "", which indicates to use the same host.
        SSLProxyHeaders:        map[string]string{"X-Forwarded-
Proto": "https"},

        // SSLProxyHeaders is set of header keys with associated
 values
        //that would indicate a
        //valid HTTPS request. Useful when using Nginx:
        //`map[string]string{"X-Forwarded-
        //Proto": "https"}`. Default is blank map.
        STSSeconds:             315360000,


        // STSSeconds is the max-age of the Strict-Transport-Sec
urity header.
        //Default is 0, which would NOT include the header.
        STSIncludeSubdomains:   true,


        // If STSIncludeSubdomains is set to true,
        //the `includeSubdomains`
        //will be appended to the Strict-Transport-Security head
er. Default is false.
        STSPreload:             true,

        // If STSPreload is set to true, the `preload`
        //flag will be appended to the Strict-Transport-Security
 header.
        //Default is false.
        ForceSTSHeader:         false,

        // STS header is only included when the connection is HT
TPS.
        //If you want to force it to always be added, set to tru
e.
```

```
        //`IsDevelopment` still overrides this. Default is false.

        FrameDeny:              true,
        // If FrameDeny is set to true, adds the X-Frame-Options
 header with
        //the value of `DENY`. Default is false.
        CustomFrameOptionsValue: "SAMEORIGIN",
        // CustomFrameOptionsValue allows the X-Frame-Options he
ader
        //value to be set with
        //a custom value. This overrides the FrameDeny option.
        ContentTypeNosniff:     true,
        // If ContentTypeNosniff is true, adds the X-Content-Typ
e-Options
        //header with the value `nosniff`. Default is false.
        BrowserXSSFilter:       true,
        // If BrowserXssFilter is true, adds the X-XSS-Protectio
n header
        //with the value `1;mode=block`. Default is false.
        ContentSecurityPolicy:  "default-src 'self'",
        // ContentSecurityPolicy allows the Content-Security-Pol
icy
        //header value to be set with a custom value. Default is
 "".
        PublicKey:              `pin-sha256="base64+primary==";
 pin-sha256="base64+backup=="; max-age=5184000; includeSubdomain
s; report-uri="https://www.example.com/hpkp-report"`,
        // PublicKey implements HPKP to prevent
        //MITM attacks with forged certificates. Default is "".

        IsDevelopment: true,
        // This will cause the AllowedHosts, SSLRedirect,
        //..and STSSeconds/STSIncludeSubdomains options to be
        //ignored during development.
        //When deploying to production, be sure to set this to f
alse.
    })

    iris.UseFunc(func(c *iris.Context) {
        err := s.Process(c)
```

```
        // If there was an error, do not continue.
        if err != nil {
            return
        }

        c.Next()
    })

    iris.Get("/home", func(c *iris.Context) {
        c.Write("Hello from /home")
    })

    iris.Listen(":8080")
}
```

Sessions

If you notice a bug or issue post it here

- Cleans the temp memory when a sessions is iddle, and re-allocate it , fast, to the temp memory when it's necessary. Also most used/regular sessions are going front in the memory's list.
- Supports any type of database, currently only redis.

A session can be defined as a server-side storage of information that is desired to persist throughout the user's interaction with the web site or web application.

Instead of storing large and constantly changing information via cookies in the user's browser, **only a unique identifier is stored on the client side** (called a "session id"). This session id is passed to the web server every time the browser makes an HTTP request (ie a page link or AJAX request). The web application pairs this session id with it's internal database/memory and retrieves the stored variables for use by the requested page.

You will see two different ways to use the sessions, I'm using the first. No performance differences.

How to use

```
package main

import    "github.com/kataras/iris"

func main() {

    /*  These are the optionally fields to configurate the sessi
ons, using the station's Config field (iris.Config.Sessions)
```

```
    // Cookie string, the session's client cookie name, for exam
ple: "irissessionid"
    Cookie string
    // DecodeCookie set it to true to decode the cookie key with
 base64 URLEncoding
    // Defaults to false
    DecodeCookie bool
    // Expires the duration of which the cookie must expires (cr
eated_time.Add(Expires)).
    // Default infinitive/unlimited life duration(0)
    Expires time.Duration
    // GcDuration every how much duration(GcDuration) the memory
 should be clear for unused cookies (GcDuration)
    // for example: time.Duration(2)*time.Hour. it will check ev
ery 2 hours if cookie hasn't be used for 2 hours,
    // deletes it from backend memory until the user comes back,
 then the session continue to work as it was
    //
    // Default 2 hours
    GcDuration time.Duration
    // DisableSubdomainPersistence set it to true in order dissa
llow your iris subdomains to have access to the session cookie
    // defaults to false
    DisableSubdomainPersistence bool
    */
    iris.Get("/", func(c *iris.Context) {
        c.Write("You should navigate to the /set, /get, /delete,
 /clear,/destroy instead")
    })
    iris.Get("/set", func(c *iris.Context) {

        //set session values
        c.Session().Set("name", "iris")

        //test if setted here
        c.Write("All ok session setted to: %s", c.Session().GetS
tring("name"))
    })

    iris.Get("/get", func(c *iris.Context) {
```

```
        // get a specific key, as string, if no found returns ju
st an empty string
        name := c.Session().GetString("name")

        c.Write("The name on the /set was: %s", name)
    })

    iris.Get("/delete", func(c *iris.Context) {
        // delete a specific key
        c.Session().Delete("name")
    })

    iris.Get("/clear", func(c *iris.Context) {
        // removes all entries
        c.Session().Clear()
    })

    iris.Get("/destroy", func(c *iris.Context) {
        //destroy, removes the entire session and cookie
        c.SessionDestroy()
        c.Log("You have to refresh the page to completely remove
 the session (on browsers), so the name should NOT be empty NOW,
 is it?\n ame: %s\n\nAlso check your cookies in your browser's c
ookies, should be no field for localhost/127.0.0.1 (or what ever
 you use)", c.Session().GetString("name"))
        c.Write("You have to refresh the page to completely remo
ve the session (on browsers), so the name should NOT be empty NO
W, is it?\nName: %s\n\nAlso check your cookies in your browser's
 cookies, should be no field for localhost/127.0.0.1 (or what ev
er you use)", c.Session().GetString("name"))
    })

    iris.Listen(":8080")
    //iris.ListenTLS("0.0.0.0:443", "mycert.cert", "mykey.key")
}
```

Example with **redis session database**, which located here

```
package main
```

```
import (
    "github.com/iris-contrib/sessiondb/redis"
    "github.com/iris-contrib/sessiondb/redis/service"
    "github.com/kataras/iris"
)

func main() {
    db := redis.New(service.Config{Network: service.DefaultRedis
Network,
        Addr:          service.DefaultRedisAddr,
        Password:      "",
        Database:      "",
        MaxIdle:       0,
        MaxActive:     0,
        IdleTimeout:   service.DefaultRedisIdleTimeout,
        Prefix:        "",
        MaxAgeSeconds: service.DefaultRedisMaxAgeSeconds}) // op
tionally configure the bridge between your redis server

    iris.UseSessionDB(db)

    iris.Get("/set", func(c *iris.Context) {

        //set session values
        c.Session().Set("name", "iris")

        //test if setted here
        c.Write("All ok session setted to: %s", c.Session().GetS
tring("name"))
    })

    iris.Get("/get", func(c *iris.Context) {
        // get a specific key, as string, if no found returns ju
st an empty string
        name := c.Session().GetString("name")

        c.Write("The name on the /set was: %s", name)
    })
```

```
    iris.Get("/delete", func(c *iris.Context) {
        // delete a specific key
        c.Session().Delete("name")
    })

    iris.Get("/clear", func(c *iris.Context) {
        // removes all entries
        c.Session().Clear()
    })

    iris.Get("/destroy", func(c *iris.Context) {
        //destroy, removes the entire session and cookie
        c.SessionDestroy()
        c.Log("You have to refresh the page to completely remove
 the session (on browsers), so the name should NOT be empty NOW,
 is it?\n ame: %s\n\nAlso check your cookies in your browser's c
ookies, should be no field for localhost/127.0.0.1 (or what ever
 you use)", c.Session().GetString("name"))
        c.Write("You have to refresh the page to completely remo
ve the session (on browsers), so the name should NOT be empty NO
W, is it?\nName: %s\n\nAlso check your cookies in your browser's
 cookies, should be no field for localhost/127.0.0.1 (or what ev
er you use)", c.Session().GetString("name"))

    })

    iris.Listen(":8080")
}
```

Websockets

WebSocket is a protocol providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C.

WebSocket is designed to be implemented in web browsers and web servers, but it can be used by any client or server application. The WebSocket Protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request. The WebSocket protocol makes more interaction between a browser and a website possible, **facilitating the real-time data transfer from and to the server**.

Read more about Websockets via wikipedia

Configuration

```
type Websocket struct {
    // WriteTimeout time allowed to write a message to the conne
ction.
    // Default value is 15 * time.Second
    WriteTimeout time.Duration
    // PongTimeout allowed to read the next pong message from th
e connection
    // Default value is 60 * time.Second
    PongTimeout time.Duration
    // PingPeriod send ping messages to the connection with this
 period. Must be less than PongTimeout
    // Default value is (PongTimeout * 9) / 10
    PingPeriod time.Duration
    // MaxMessageSize max message size allowed from connection
    // Default value is 1024
    MaxMessageSize int64
    // Endpoint is the path which the websocket server will list
en for clients/connections
    // Default value is empty string, if you don't set it the We
bsocket server is disabled.
    Endpoint string
    // Headers  the response headers before upgrader
    // Default is empty
    Headers map[string]string
    // ReadBufferSize is the buffer size for the underline reader

    ReadBufferSize int
    // WriteBufferSize is the buffer size for the underline writ
er
    WriteBufferSize int
}
```

```
iris.Config.Websocket.Endpoint = "/myEndpoint"
```

Outline

```
iris.Websocket.OnConnection(func(c iris.WebsocketConnection){})
```

WebsocketConnection's methods

```
// Receive from the client
On("anyCustomEvent", func(message string) {})
On("anyCustomEvent", func(message int){})
On("anyCustomEvent", func(message bool){})
On("anyCustomEvent", func(message anyCustomType){})
On("anyCustomEvent", func(){})

// Receive a native websocket message from the client
// compatible without need of import the iris-ws.js to the .html
OnMessage(func(message []byte){})

// Send to the client
Emit("anyCustomEvent", string)
Emit("anyCustomEvent", int)
Emit("anyCustomEvent", bool)
Emit("anyCustomEvent", anyCustomType)

// Send via native websocket way, compatible without need of imp
ort the iris-ws.js to the .html
EmitMessage([]byte("anyMessage"))

// Send to specific client(s)
To("otherConnectionId").Emit/EmitMessage...
To("anyCustomRoom").Emit/EmitMessage...

// Send to all opened connections/clients
To(websocket.All).Emit/EmitMessage...

// Send to all opened connections/clients EXCEPT this client(c)
To(websocket.NotMe).Emit/EmitMessage...

// Rooms, group of connections/clients
Join("anyCustomRoom")
Leave("anyCustomRoom")


// Fired when the connection is closed
OnDisconnect(func(){})
```

How to use

Server-side

```
// ./main.go
package main

import (
    "fmt"

    "github.com/kataras/iris"
)

type clientPage struct {
    Title string
    Host  string
}

func main() {
    iris.Static("/js", "./static/js", 1)

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("client.html", clientPage{"Client Page", ctx.
HostString()})
    })

    // the path which the websocket client should listen/registe
d to ->
    iris.Config.Websocket.Endpoint = "/my_endpoint"
    // for Allow origin you can make use of the middleware
    //iris.Config().Websocket.Headers["Access-Control-Allow-Orig
in"] = "*"

    var myChatRoom = "room1"
    iris.Websocket.OnConnection(func(c iris.WebsocketConnection)
 {

        c.Join(myChatRoom)
```

```
        c.On("chat", func(message string) {
            // to all except this connection ->
            //c.To(websocket.Broadcast).Emit("chat", "Message fr
om: "+c.ID()+"-> "+message)

            // to the client ->
            //c.Emit("chat", "Message from myself: "+message)

            //send the message to the whole room,
            //all connections are inside this room will receive
this message
            c.To(myChatRoom).Emit("chat", "From: "+c.ID()+": "+m
essage)
        })

        c.OnDisconnect(func() {
            fmt.Printf("\nConnection with ID: %s has been discon
nected!", c.ID())
        })
    })

    iris.Listen(":8080")
}
```

Client-side

```
// js/chat.js
var messageTxt;
var messages;

$(function () {

    messageTxt = $("#messageTxt");
    messages = $("#messages");


    ws = new Ws("ws://" + HOST + "/my_endpoint");
    ws.OnConnect(function () {
        console.log("Websocket connection enstablished");
```

```
    });

    ws.OnDisconnect(function () {
        appendMessage($("<div><center><h3>Disconnected</h3></cen
ter></div>"));
    });

    ws.On("chat", function (message) {
        appendMessage($("<div>" + message + "</div>"));
    })

    $("#sendBtn").click(function () {
        //ws.EmitMessage(messageTxt.val());
        ws.Emit("chat", messageTxt.val().toString());
        messageTxt.val("");
    })

})


function appendMessage(messageDiv) {
    var theDiv = messages[0]
    var doScroll = theDiv.scrollTop == theDiv.scrollHeight - the
Div.clientHeight;
    messageDiv.appendTo(messages)
    if (doScroll) {
        theDiv.scrollTop = theDiv.scrollHeight - theDiv.clientHe
ight;
    }
}
```

```
<html>

<head>
    <title>My iris-ws</title>
</head>

<body>
    <div id="messages" style="border-width:1px;border-style:soli
d;height:400px;width:375px;">

    </div>
    <input type="text" id="messageTxt" />
    <button type="button" id="sendBtn">Send</button>
    <script type="text/javascript">
        var HOST = {{.Host}}
    </script>
    <script src="js/vendor/jquery-2.2.3.min.js" type="text/javas
cript"></script>
    <!-- /iris-ws.js is served automatically by the server -->
    <script src="/iris-ws.js" type="text/javascript"></script>
    <!-- -->
    <script src="js/chat.js" type="text/javascript"></script>
</body>

</html>
```

View a working example by navigating here and if you need more than one websocket server click here

Graceful

This is a package.

Enables graceful shutdown.

```
package main

import (
    "time"
        "github.com/kataras/iris"
    "github.com/iris-contrib/graceful"
)

func main() {
    api := iris.New()
    api.Get("/", func(c *iris.Context) {
        c.Write("Welcome to the home page!")
    })

    graceful.Run(":3001", time.Duration(10)*time.Second, api)
}
```

Recovery

This is a middleware.

Safety recover the server from panic.

```
recovery.New(...*logger.Logger)
```

```go

package main

import ( "github.com/kataras/iris" "github.com/iris-contrib/middleware/recovery" )

func main() {

iris.Use(recovery.New(iris.Logger)) // optional parameter is the
 logger which the stack of the panic will be printed, here we're
 using the default station's Logger.

iris.Get("/", func(ctx *iris.Context) {
    ctx.Write("Hi, let's panic")
    panic("errorrrrrrrrrrrrrr")
})

iris.Listen(":8080")
```

}

Plugins

Plugins are modules that you can build to inject the Iris' flow. Think it like a middleware for the Iris framework itself, not the requests. Middleware starts it's actions after the server listen and executes itself on every request, Plugin on the other hand starts working when you registered it, it has to do with framework's code, it has access to the *iris.Framework, so it can register routes, start a second server, read the iris' configs or edit them and all things you can do with Iris. Look how it's interface looks:

```
type (
    // Plugin just an empty base for plugins
    // A Plugin can be added with: .Add(PreListenFunc(func(*Fram
ework))) and so on... or
    // .Add(myPlugin{},myPlugin2{}) which myPlugin is  a struct
with any of the methods below or
    //.PostListen(func(*Framework)) and so on...
    Plugin interface {
    }

    // pluginGetName implements the GetName() string method
    pluginGetName interface {
        // GetName has to returns the name of the plugin, a name
 is unique
        // name has to be not dependent from other methods of th
e plugin,
        // because it is being called even before the Activate
        GetName() string
    }

    // pluginGetDescription implements the GetDescription() stri
ng method
    pluginGetDescription interface {
        // GetDescription has to returns the description of what
 the plugins is used for
        GetDescription() string
    }
```

```
    // pluginActivate implements the Activate(pluginContainer) e
rror method
    pluginActivate interface {
        // Activate called BEFORE the plugin being added to the
plugins list,
        // if Activate returns none nil error then the plugin is
 not being added to the list
        // it is being called only one time
        //
        // PluginContainer parameter used to add other plugins i
f that's necessary by the plugin
        Activate(PluginContainer) error
    }
    // pluginPreListen implements the PreListen(*Framework) meth
od
    pluginPreListen interface {
        // PreListen it's being called only one time, BEFORE the
 Server is started (if .Listen called)
        // is used to do work at the time all other things are r
eady to go
        //  parameter is the station
        PreListen(*Framework)
    }
    // PreListenFunc implements the simple function listener for
 the PreListen(*Framework)
    PreListenFunc func(*Framework)
    // pluginPostListen implements the PostListen(*Framework) me
thod
    pluginPostListen interface {
        // PostListen it's being called only one time, AFTER the
 Server is started (if .Listen called)
        // parameter is the station
        PostListen(*Framework)
    }
    // PostListenFunc implements the simple function listener fo
r the PostListen(*Framework)
    PostListenFunc func(*Framework)
    // pluginPreClose implements the PreClose(*Framework) method
    pluginPreClose interface {
```

```
        // PreClose it's being called only one time, BEFORE the
Iris .Close method
        // any plugin cleanup/clear memory happens here
        //
        // The plugin is deactivated after this state
        PreClose(*Framework)
    }
    // PreCloseFunc implements the simple function listener for
the PreClose(*Framework)
    PreCloseFunc func(*Framework)

    // pluginPreDownload It's for the future, not being used, I
need to create
    // and return an ActivatedPlugin type which will have it's m
ethods, and pass it on .Activate
    // but now we return the whole pluginContainer, which I can'
t determinate which plugin tries to
    // download something, so we will leave it here for the futu
re.
    pluginPreDownload interface {
        // PreDownload it's being called every time a plugin tri
es to download something
        //
        // first parameter is the plugin
        // second parameter is the download url
        // must return a boolean, if false then the plugin is no
t permmited to download this file
        PreDownload(plugin Plugin, downloadURL string) // bool
    }

    // PreDownloadFunc implements the simple function listener f
or the PreDownload(plugin,string)
    PreDownloadFunc func(Plugin, string)

)
```

```
package main

import (
```

```
    "fmt"

    "github.com/kataras/iris"
)

func main() {
    // first way:
    // simple way for simple things
    // PreListen before a station is listening ( iris.Listen/TLS
...)
    iris.Plugins.PreListen(func(s *iris.Framework) {
        for _, route := range s.Lookups() {
            fmt.Printf("Func: Route Method: %s | Subdomain %s |
Path: %s is going to be registed with %d handler(s). \n", route.
Method(), route.Subdomain(), route.Path(), len(route.Middleware(
)))
        }
    })

    // second way:
    // structured way for more things
    plugin := myPlugin{}
    iris.Plugins.Add(plugin)

    iris.Get("/first_route", aHandler)

    iris.Post("/second_route", aHandler)

    iris.Put("/third_route", aHandler)

    iris.Get("/fourth_route", aHandler)

    iris.Listen(":8080")
}

func aHandler(ctx *iris.Context) {
    ctx.Write("Hello from: %s", ctx.PathString())
}

type myPlugin struct{}
```

```
// PostListen after a station is listening ( iris.Listen/TLS...)
func (pl myPlugin) PostListen(s *iris.Framework) {
    fmt.Printf("myPlugin: server is listening on host: %s", s.HT
TPServer.Host())
}

//list:
/*
    Activate(iris.PluginContainer)
    GetName() string
    GetDescription() string
    PreListen(*iris.Framework)
    PostListen(*iris.Framework)
    PreClose(*iris.Framework)
    PreDownload(thePlugin iris.Plugin, downloadUrl string)
    // for custom events
    On(string,...func())
    Call(string)
*/
```

An example of one plugin which is under development is the Iris control, a web interface that gives you control to your server remotely. You can find it's code here.

Take a look at the real plugins, easy to make your own.

Internationalization and Localization

This is a middleware

Tutorial

Create folder named 'locales'

```
///Files:

./locales/locale_en-US.ini
./locales/locale_el-US.ini
```

Contents on locale_en-US:

```
hi = hello, %s
```

Contents on locale_el-GR:

```
hi = Γειά, %s
```

```
    package main

    import (
        "fmt"
        "github.com/kataras/iris"
        "github.com/iris-contrib/middleware/i18n"
    )

    func main() {

        iris.Use(i18n.New(i18n.Config{Default: "en-US",
            Languages: map[string]string{
                "en-US": "./locales/locale_en-US.ini",
                "el-GR": "./locales/locale_el-GR.ini",
                "zh-CN": "./locales/locale_zh-CN.ini"}}))

        iris.Get("/", func(ctx *iris.Context) {
            hi := ctx.GetFmt("translate")("hi", "maki") // hi is
 the key, 'maki' is the %s, the second parameter is optional
            language := ctx.Get("language") // language is the l
anguage key, example 'en-US'

            ctx.Write("From the language %s translated output: %
s", language, hi)
        })

        iris.Listen(":8080")

    }
```

Typescript

This is a plugin.

This is an Iris and typescript bridge plugin.

What?

1. Search for typescript files (.ts)
2. Search for typescript projects (.tsconfig)
3. If 1 || 2 continue else stop
4. Check if typescript is installed, if not then auto-install it (always inside npm global modules, -g)
5. If typescript project then build the project using tsc -p $dir
6. If typescript files and no project then build each typescript using tsc $filename
7. Watch typescript files if any changes happens, then re-build (5|6)

> Note: Ignore all typescript files & projects whose path has '/node_modules/'

Options

- **Bin**: string, the typescript installation path/bin/tsc or tsc.cmd, if empty then it will search to the global npm modules
- **Dir**: string, Dir set the root, where to search for typescript files/project. Default "./"
- **Ignore**: string, comma separated ignore typescript files/project from these directories. Default "" (node_modules are always ignored)
- **Tsconfig**: config.Tsconfig{}, here you can set all compilerOptions if no tsconfig.json exists inside the 'Dir'
- **Editor**: config.Typescript { Editor: config.Editor{}, if setted then alm-tools browser-based typescript IDE will be available. Defailt is nil

> All these are optional

How to use

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/plugin/typescript"
)

func main(){
    ts := typescript.Config {
        Dir: "./scripts/src",
        Tsconfig: typescript.Tsconfig{Module: "commonjs", Target
: "es5"},
    }
    // or typescript.DefaultConfig()

    iris.Plugins.Add(typescript.New(ts)) //or with the default o
ptions just: typescript.New()

    iris.Get("/", func (ctx *iris.Context){})

    iris.Listen(":8080")
}
```

Enable web browser editor

```
ts := typescript.Typescript {
    //...
    Editor: typescript.Editor{Username:"admin", Password: "admin
!123"}
    //...
}
```

Editor

This is a plugin.

Editor Plugin is just a bridge between Iris and alm-tools.

alm-tools is a typescript online IDE/Editor, made by @basarat one of the top contributors of the Typescript.

Iris gives you the opportunity to edit your client-side using the alm-tools editor, via the editor plugin.

This plugin starts it's own server, if Iris server is using TLS then the editor will use the same key and cert.

How to use

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/plugin/editor"
)

func main(){
    e := editor.New()
   // editor.Config{ Username: "admin", Password: "admin!123", P
ort: 4444, WorkingDir: "/public/scripts"}

    iris.Plugins.Add(e)

    iris.Get("/", func (ctx *iris.Context){})

    iris.Listen(":8080")
}
```

Note for username, password: The Authorization specifies the authentication mechanism (in this case Basic) followed by the username and password. Although, the string aHR0cHdhdGNoOmY= may look encrypted it is simply a base64 encoded version of username:password. Would be readily available to anyone who could intercept the HTTP request. Read more here.

> The editor can't work if the directory doesn't contains a tsconfig.json.
>
> If you are using the typescript plugin you don't have to call the .Dir(...)

Control panel

This is a plugin which is working but not finished yet.

Which gives access to your iris server's information via a web interface.

> You need internet connection the first time you will run this plugin, because the assets don't exists to this repository but here. The plugin will install these for you at the first run.

How to use

```
iriscontrol.New(port int, authenticatedUsers map[string]string)
iris.IPlugin
```

Example

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/plugin/iriscontrol"
)

func main() {

    iris.Plugins.Add(iriscontrol.New(9090, map[string]string{
        "irisusername1": "irispassword1",
        "irisusername2": "irispassowrd2",
    }))
    //or
    // ....
    // iriscontrol.New(iriscontrol.Config{...})

    iris.Get("/", func(ctx *iris.Context) {
    })

    iris.Post("/something", func(ctx *iris.Context) {
    })

    iris.Listen(":8080")
}
```