PRACTICAL GUIDE [THE DONATOR EDITION]

IRIS WEB FRAMEWORK

BY GERASIMOS MAROPOULOS
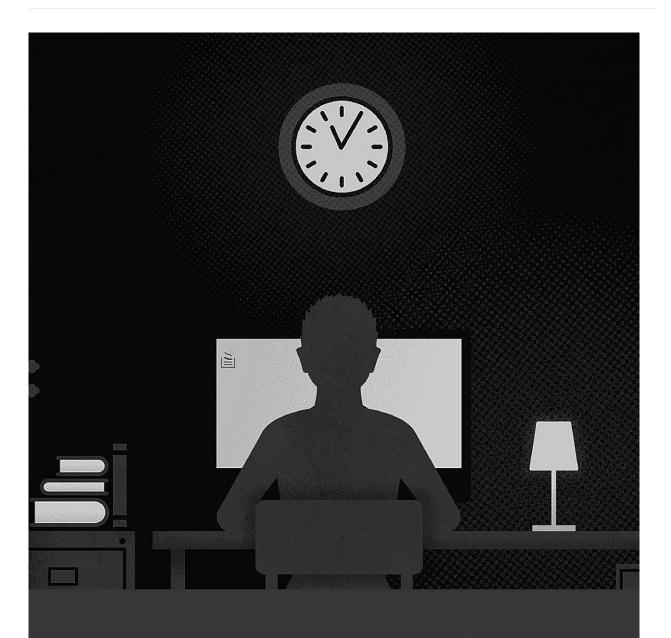
#The Fastest Web Framework

Table of Contents

Table of Contents

PRACTICAL GUIDE **[THE SPECIAL EDITION]**

IRIS WEB FRAMEWORK

BY GERASIMOS MAROPOULOS

#The Fastest Web Framework

Table of Contents

- 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

> NOTE: If you want to view the docs for the most recent stable version, v4 LTS, click here instead

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

The results showed that the req / sec golang-iris do best at around 70k-50k, followed by nginx and nginx-php-fpm and nodejs respectively.
The error golang-iris and nginx work equally, followed by the final nginx and php-fpm at a ratio of 1: 1.
For the test, it is still uncertain. Inaccurate, but it is in an acceptable range.
The computer is used for spec

ubuntu 16.04 LTS

Device name	eazy
Memory	7.7 GiB
Processor	Intel® Core™ i5-4210U CPU @ 1.70GHz × 4
Graphics	Intel® Haswell Mobile
OS type	64-bit
Disk	125.9 GB

FROM IRIS vs NGINX vs NGINX-PHP vs NODEJS Express
https://www.ntossapo.me/2016/08/13/nginx-vs-nginx-php-fpm-vs-go-iris-vs-express-with-wrk/

GLUG #11

f Tell a friend   🐦 Share

🕐 **Thursday, October 13, 2016**
7:00 PM

📍 **Biuro Young/Skilled**
Kuźnicza 10, Wroclaw (map)

4th floor

Beginners and experts (both) see you all in October!

• **(B)** Iris Web Framework (PS, 30mins)

• **(B)** Lessons learned based on mobile game backend (MD, 30mins)

https://hackpad.com/G.L.U.G.-Wroclaw-Xzzvs7NV5bP

Join or login to comment.

Pawel K.
Attention: new venue!
September 21

Wroclaw, Poland
Founded Sep 20, 2015

About us...

Gophers	191
Group reviews	2
Upcoming Meetups	1
Past Meetups	12
Our calendar	📅

Organizers:
Mike, Maciej Talaśka, Mateusz Dymiński, Pawel Kowalak

✉ Contact

Want to go?

Join and RSVP

30 going

Mateusz D.
Assistant Organizer, Event Host

Pawel K.
Co-Organizer, Event Host

Michał M.

Jakub B.

Damian R.

Jan

Olek P.

Hacker News
@newsycombinator

👤 Follow

Iris: Fast back-end web framework for Go iris-go.com

RETWEETS	LIKES
20	61

11:01 PM - 20 Jun 2016

↩   ♻ 20   ♥ 61   •••

HN Forum Link Bot @hntlbot · Jun 20
.@newsycombinator Forum link: news.ycombinator.com/item?id=119436...

↩   ♻   ♥   •••

🐦

qskousen commented 16 days ago   +😊 ✏ ✕

Iris should definitely stick with the Iris goddess meaning, and here's why:

- It was **@kataras** intention when he named the framework in the first place.

- Iris the goddess is the "personification of the rainbow and messenger of the gods" and Iris brings many things together into one (like a rainbow brings colors together) and sends messages back and froth between server and client, as Iris carries messages between the gods and mortals.

- Iris "travels with the speed of wind from one end of the world to the other", and Iris is the fastest web framework.

- "As a goddess, Iris is associated with communication, messages, the rainbow and new endeavors." I think the parallels in that to Iris framework are pretty clear.

- Iris the goddess has golden wings. I don't know how that relates to Iris the framework, but it's pretty awesome.

David V. Wallin
@DvWallin   ⚙ Ακολουθείτε

Gold stars to the incredible developer of Iris - @MakisMaropoulos - for being the most dedicated FOSS developer I've seen of late. #golang

Jonathan Dion
@_jondion   ⚙ 👤 Ακολουθήστε

@MakisMaropoulos thanks for Iris, finally a good framework for Go.

Go Time
@GoTimeFM

⚙ ♟ Ακολουθήστε

Via @carlisia "Yet another (fast) web framework (YAWF)" for Go called Iris from @MakisMaropoulos chlg.co/1ZBWiK7 #golang #gotime9
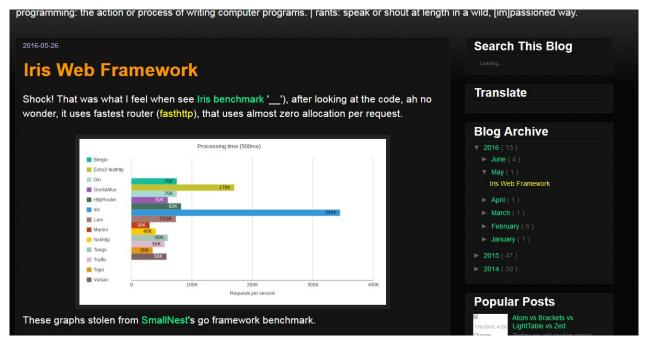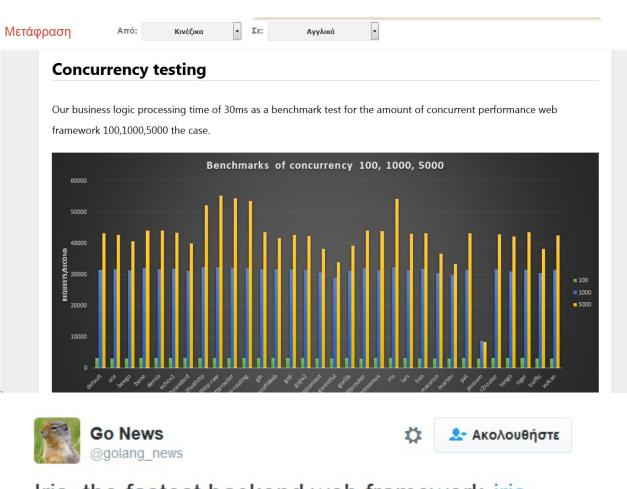
🌐 Προβολή μετάφρασης

kataras/iris

iris - Fast, unopinionated, minimalist web framework for Go. Built on top of fasthttp, up to 20x faster than the rest.

github.com

programming: the action or process of writing computer programs. | rants: speak or shout at length in a wild, [im]passioned way.

2016-05-26

Iris Web Framework

Shock! That was what I feel when see Iris benchmark '__'), after looking at the code, ah no wonder, it uses fastest router (fasthttp), that uses almost zero allocation per request.

Processing time (500ms)

These graphs stolen from SmallNest's go framework benchmark.

Search This Blog

Loading...

Translate

Blog Archive

▼ 2016 (13)
  ► June (4)
  ▼ May (1)
    Iris Web Framework
  ► April (1)
  ► March (1)
  ► February (5)
  ► January (1)
► 2015 (47)
► 2014 (39)

Popular Posts

Atom vs Brackets vs LightTable vs Zed
7/9/2015, 4:29:

Concurrency testing

Our business logic processing time of 30ms as a benchmark test for the amount of concurrent performance web framework 100,1000,5000 the case.

Benchmarks of concurrency 100, 1000, 5000

Go News
@golang_news

⚙   👤+ Ακολουθήστε

Iris, the fastest backend web framework iris-go.com #reddit

🌐 Προβολή μετάφρασης

RETWEETS	ΑΡΕΣΕΙ ΣΕ	
2	8	

9:32 μ.μ. - 17 Ιουν 2016

↩   ⇄ 2   ♥ 8   •••

Klaus Post ʕ◔ϖ◔ʔ
@sh0dan

⚙ 👤⁺ Ακολουθήστε

Very impressive stuff from @MakisMaropoulos - will be interesting to try out and follow!

> **Go News** @golang_news
> Iris, the fastest backend web framework iris-go.com #reddit

🌐 Προβολή μετάφρασης

1
ΑΡΕΣΕΙ ΣΕ

9:51 μ.μ. - 17 Ιουν 2016

↩   ♺   ♥ 1   •••

Michael Herman
@MikeHerman

⚙ 👤⁺ Ακολουθήστε

Iris - The fastest backend web framework for Go >> iris-go.com by @MakisMaropoulos #golang #webdevelopment

🌐 Προβολή μετάφρασης

ΑΡΕΣΕΙ ΣΕ
7

11:51 π.μ. - 21 Ιουν 2016

↩   ♺   ♥ 7   •••

Manigandan
@ManigandanJeff

⚙ 👤 Ακολουθήστε

really its fastest in the world :p have to try it out once

> **Makis Maropoulos** @MakisMaropoulos
>
> #golang #iris is first on github go trends and 4th on all languages, thanks goes to all of you!!

🌐 Προβολή μετάφρασης

4:00 μ.μ. - 21 Ιουν 2016

prithviraj sukale
@pvsukale

⚙ 👤 Ακολουθήστε

@MakisMaropoulos thanks for creating iris !

🌐 Προβολή μετάφρασης

9:51 μ.μ. - 21 Ιουν 2016

Beard of War
@blainsmith

⚙ 👤 Ακολουθήστε

The speed looks impressive for Iris iris-go.com @MakisMaropoulos #golang

🌐 Προβολή μετάφρασης

10:30 μ.μ. - 21 Ιουν 2016

📍 Saratoga Springs, NY

Etienne Bruines @EtienneBruines   16:26
Have been checking out new software for the last 6 years or so, never was anything faster than nginx (static files)

Vegax @vegax87   16:26
IS this the beginning of the end of nginx?

Steve High
@evilnode

⚙ 👤 Ακολουθήστε

Wow. @MakisMaropoulos … #iris is looking really, really good. Great work!

🌐 Προβολή μετάφρασης

4:00 μ.μ. - 22 Ιουν 2016

Bob Hannent
@bobdvb

⚙ 👤 Ακολουθήστε

Go #Greece! @MakisMaropoulos
RT @bytemark gitbook.com/book/kataras/i…
"the fastest web framework for Go" -
impressive for 3 months work ^M

🌐 Προβολή μετάφρασης

GitBook · Writing made easy
GitBook is where you create, write and organize documentation and books with your team.
gitbook.com

5:23 μ.μ. - 22 Ιουν 2016

omgj @omgj   Jun 23 13:26 ✓ ⋯
@kataras still trying to wrap my head around the whole thing. Can't believe you did this by yourself

Srinath @srinathgs   Jun 23 13:30
  @kataras still trying to wrap my head around the whole thing. Can't believe you did this by yourself - Exactly my feelings about Iris

Personal   Open source   Business   Explore

Pricing   Blog   Support

Search GitHub

Your dashboard

Explore GitHub

Showcases   Integrations   Trending   Stars

Trending in open source

See what the GitHub community is most excited about this month.

Repositories   Developers

Trending: **this month** ▾

All languages

Unknown languages

Go

Other: **Languages** ▾

ProTip! Looking for recently updated Go repositories? Try this search

kataras/**iris**   ★ Star

The fastest web framework for Go in (THIS) earth

Go • 2,119 stars this month • Built by 👤

getlantern/**lantern**   ★ Star

⚡ Open Internet for everyone. Lantern is a free desktop application that delivers fast, reliable and secure access to the open Internet for users in censored regions. It uses a variety of techniques to stay unblocked, including P2P and domain fronting. Lantern relies on users in uncensored regions acting as access points to the open Internet.

Go • 1,777 stars this month • Built by

coreos/**torus**   ★ Star

Torus Distributed Storage

Go • 1,210 stars this month • Built by

I'll look at iris.StaticContent, thank you and great work!

BO   **boolow5** 4:24 AM
         Has joined the channel.

BO   **boolow5** 4:26 AM
         @kataras: Thank you very much for build such a useful framework.

ivan.cduran 4:27 AM
in the company where I work we started using iris to create a large project for a television network

probably about 10 people are using iris here to the development of that project

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 inject your own plugins 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, authenticity and data integrity between your server and the client, you can serve using letsencrypt.org, automatic tls too*
- **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
- **Auto updater**: You're able to turn on the version checker & updater in case you forget to update your iris

Iris is one of the most featured web frameworks out there, not all features are here and don't expect from me to write down all of their usages in this gitbook, if you see that I'm missing something please make a PR to the gitbook repository!

Versioning

Current: **v5.0.1**

Stable: **v4 LTS**

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

The only requirement is the Go Programming Language, at least v1.7.

```
$ 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 might have problems installing Iris.
  Follow the below steps:

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

- `$ 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` .

NOTE: **If you want a stable version for production**, then install v4 instead, its examples, book, middleware and plugins are expecting an import path of:

```
$ go get -u gopkg.in/kataras/iris.v4/iris
```

Hi

```go
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:

```go
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**:

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

```go
// ./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**:

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

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

Listen & Serve functions

```
// Serve serves incoming connections from the given listener.
//
// Serve blocks until the given listener returns permanent error.

Serve(ln net.Listener) error

// 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 Serve
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 Serve
// ex: iris.ListenTLS(":8080","yourfile.cert","yourfile.key")
ListenTLS(addr string, certFile, keyFile string)

// ListenLETSENCRYPT starts a server listening at the specific n
at address
// using key & certification taken from the letsencrypt.org 's s
ervers
// it's also starts a second 'http' server to redirect all 'http
://$ADDR_HOSTNAME:80' to the' https://$ADDR'
// example: https://github.com/iris-contrib/examples/blob/master
/letsencrypt/main.go
ListenLETSENCRYPT(addr string)
```

```go
// 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 Serve
// ex: iris.ListenUNIX(":8080", Mode: os.FileMode)
ListenUNIX(addr string, mode os.FileMode)


// 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



// Reserve re-starts the server using the last .Serve's listener
Reserve() error

// IsRunning returns true if server is running
IsRunning() bool
```

```go
// Serve
ln, err := net.Listen("tcp4", ":8080")
if err := iris.Serve(ln); err != nil {
    panic(err)
}
// same as api := iris.New(); api.Serve(ln), iris. contains a de
fault iris instance, this exists for
// any function or field you will see at the rest of the gitbook.


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

// ListenTLS
iris.ListenTLS(":8080", "./ssl/mycert.cert", "./ssl/mykey.key")

// and so on...
```

```go
// Package main provide one-line integration with letsencrypt.org

package main

import "github.com/kataras/iris"

func main() {
    iris.Get("/", func(ctx *iris.Context) {
        ctx.Write("Hello from SECURE SERVER!")
    })

    iris.Get("/test2", func(ctx *iris.Context) {
        ctx.Write("Welcome to secure server from /test2!")
    })

    // This will provide you automatic certification & key from
letsencrypt.org's servers
    // it also starts a second 'http://' server which will redir
ect all 'http://$PATH'
    // requests to 'https://$PATH'
    iris.ListenLETSENCRYPT("127.0.0.1:443")
}
```

Examples:

- Listen using a custom fasthttp server
- Serve content using a custom router
- Listen using a custom net.Listener
- Redirect all http://$HOST to https://$HOST using a custom net.Listener
- Listen using automatic ssl

Handlers

Handlers, as the name implies, handle requests.
Each of the handler registration methods described in the following subchapters
returns a `RouteNameFunc` type.

Handlers must implement the Handler interface:

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

Once the handler is registered, we can use the returned `RouteNameFunc` type
(which is actually just a `func` type) to give a name to the handler registration for
easier lookup in code or in templates. For more information, checkout the Routing
and reverse lookups section.

Using Handlers

```go
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 the most simple method to register a route or a middleware, but under the hood it acts like a Handler. It 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 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{}))
```

Routing

As mentioned in the Handlers chapter, Iris provides several handler registration methods, each of which returns a `RouteNameFunc` type.

Route naming

Route naming is easy, since we just call the returned `RouteNameFunc` with a string parameter to define a name:

```go
package main

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

func main() {

    // define a function
    render := func(ctx *iris.Context) {
        ctx.Render("index.html", nil)
    }

    // handler registration and naming
    iris.Get("/", render)("home")
    iris.Get("/about", render)("about")
    iris.Get("/page/:id", render)("page")

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

Route reversing AKA generating URLs from the route name

When we register the handlers for a specific path, we get the ability to create URLs based on the structured data we pass to Iris. In the example above, we've named three routers, one of which even takes parameters. If we're using the default `html/template` templating engine, we can use a simple action to reverse the routes (and generae actual URLs):

```
Home: {{ url "home" }}
About: {{ url "about" }}
Page 17: {{ url "page" "17" }}
```

Above code would generate the following output:

```
Home: http://0.0.0.0:8080/
About: http://0.0.0.0:8080/about
Page 17: http://0.0.0.0:8080/page/17
```

Using route names in code

We can use the following methods/functions to work with named routes (and their parameters):

- global `Lookups` function to get all registered routes
- `Lookup(routeName string)` framework method to retrieve a route by name
- `URL(routeName string, args ...interface{})` framework method to generate url string based on supplied parameters
- `Path(routeName string, args ...interface{}` framework method to generate just the path (without host and protocol) portion of the URL based on provided values
- `RedirectTo(routeName string, args ...interface{})` context method to return a redirect response to a URL defined by the named route and optional parameters

Examples

Check out the `template_engines/template_html_4` example in the `iris-contrib/examples` repository.

Middleware

Quick view

```go
// First mount 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){})

// declare any finish middleware/ runs always at the end of the
request using .Done/.DoneFunc
iris.DoneFunc(executeLast)

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

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



// executeLast func middleware

func executeLast(ctx *iris.Context){
    println("before close the http request")
}

// myMiddleware will be like that

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

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

Middlewares in Iris are not complicated to implement, they are similar to simple Handlers.

They implement the Handler interface as well:

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

Handler middleware example:

```go
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:

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


iris.UseFunc(myMiddleware)
```

HandlerFunc middleware for a specific route:

```go
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 middlewares must come before route declarations.

Make use of the middleware package, view practical examples here.

Middleware

```go
package main

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

type Page struct {
    Title string
}

iris.Use(logger.New())

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

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

Done\/DoneFunc

```go
package main

import "github.com/kataras/iris"

func firstMiddleware(ctx *iris.Context) {
    ctx.Write("1. This is the first middleware, before any of ro
ute handlers \n")
    ctx.Next()
}

func secondMiddleware(ctx *iris.Context) {
    ctx.Write("2. This is the second middleware, before the '/'
route handler \n")
    ctx.Next()
}

func thirdMiddleware(ctx *iris.Context) {
    ctx.Write("3. This is the 3rd middleware, after the '/' rout
```

```go
e handler \n")
    ctx.Next()
}

func lastAlwaysMiddleware(ctx *iris.Context) {
    ctx.Write("4. This is ALWAYS the last Handler \n")
}

func main() {

    iris.UseFunc(firstMiddleware)
    iris.DoneFunc(lastAlwaysMiddleware)

    iris.Get("/", secondMiddleware, func(ctx *iris.Context) {
        ctx.Write("Hello from / \n")
        ctx.Next() // .Next because we 're using the third middl
eware after that, and lastAlwaysMiddleware also
    }, thirdMiddleware)

    iris.Listen(":8080")

}
```

Done\/DoneFunc with Parties

```go
// Package main same as middleware_2 but with party
package main

import "github.com/kataras/iris"

func firstMiddleware(ctx *iris.Context) {
    ctx.Write("1. This is the first middleware, before any of ro
ute handlers \n")
    ctx.Next()
}

func secondMiddleware(ctx *iris.Context) {
    ctx.Write("2. This is the second middleware, before the '/'
route handler \n")
```

```go
    ctx.Next()
}

func thirdMiddleware(ctx *iris.Context) {
    ctx.Write("3. This is the 3rd middleware, after the '/' rout
e handler \n")
    ctx.Next()
}

func lastAlwaysMiddleware(ctx *iris.Context) {
    ctx.Write("4. This is ALWAYS the last Handler \n")
}

func main() {

    // with parties:
    myParty := iris.Party("/myparty", firstMiddleware).DoneFunc(
lastAlwaysMiddleware)
    {
        myParty.Get("/", secondMiddleware, func(ctx *iris.Contex
t) {
            ctx.Write("Hello from /myparty/ \n")
            ctx.Next() // .Next because we 're using the third m
iddleware after that, and lastAlwaysMiddleware also
        }, thirdMiddleware)

    }

    iris.Listen(":8080")

}
```

> Done\/DoneFuncs are just last-executed handlers, like Use\/UseFunc the
> children party inheritates these 'done\/last' handlers too.

API

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

```go
package main

import "github.com/kataras/iris"

func main() {
    // declare the routes
    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.Connect("/testConnect", testConnect)
    iris.Trace("/testTrace", testTrace)

    // start the server
    iris.Listen(":8080")
}

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

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

Declaration

You might have asked yourself:

- Q: Other frameworks need more lines to start a server, why is Iris different?
- A: Iris gives you the freedom to choose between four 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(iris.Configuration{...})**
  4. declare a new iris station with custom options: **api := iris.New(iris.OptionCharset("UTF-8"), iris.OptionSessionsCookie("mycookie"), ...)**

Configuration is **OPTIONAL** and can change after declaration with `$instance.Config.` , V `$instance.Set(Option...)`

```go
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")
}


// 3.
func thirdWay() {

    config := iris.Configuration{IsDevelopment: true}
```

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


// 4.
func forthWay() {

    api := iris.New()
    api.Set(iris.OptionCharset("UTF-8"))

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


// after .New, at runtime, also possible because Iris have defau
lt values, configuration is TOTALLY OPTIONAL DSESIRE
func main() {
    iris.Config.Websocket.Endpoind = "/ws"

    //...

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

Let's take a quick look at the **iris.Configuration**

```go
// Configuration the whole configuration for an iris instance ($
instance.Config) or global iris instance (iris.Config)
// these can be passed via options also, look at the top of this
 file(configuration.go)
//
// Configuration is also implements the OptionSet so it's a vali
d option itself, this is briliant enough
type Configuration struct {
    // VHost is the addr or the domain that server listens to, w
hich it's optional
    // When to set VHost manually:
    // 1. it's automatically setted when you're calling
```

```
    //      $instance.Listen/ListenUNIX/ListenTLS/ListenLETSENCRY
PT functions or
    //      ln,_ := iris.TCP4/UNIX/TLS/LETSENCRYPT; $instance.Ser
ve(ln)
    // 2. If you using a balancer, or something like nginx
    //    then set it in order to have the correct url
    //    when calling the template helper '{{url }}'
    //    *keep note that you can use {{urlpath }}) instead*
    //
    // Note: this is the main's server Host, you can setup unlim
ited number of fasthttp servers
    // listening to the $instance.Handler after the manually-cal
led $instance.Build
    //
    // Default comes from iris.Listen/.Serve with iris' listener
s (iris.TCP4/UNIX/TLS/LETSENCRYPT)
    VHost string

    // VScheme is the scheme (http:// or https://) putted at the
 template function '{{url }}'
    // It's an optional field,
    // When to set VScheme manually:
    // 1. You didn't start the main server using $instance.Liste
n/ListenTLS/ListenLETSENCRYPT or $instance.Serve($instance.TCP4(
)/.TLS...)
    // 2. if you're using something like nginx and have iris lis
tening with addr only(http://) but the nginx mapper is listening
 to https://
    //
    // Default comes from iris.Listen/.Serve with iris' listener
s (TCP4,UNIX,TLS,LETSENCRYPT)
    VScheme string

    // MaxRequestBodySize Maximum request body size.
    //
    // The server rejects requests with bodies exceeding this li
mit.
    //
    // By default request body size is 8MB.
    MaxRequestBodySize int
```

```
    // Per-connection buffer size for requests' reading.
    // This also limits the maximum header size.
    //
    // Increase this buffer if your clients send multi-KB Reques
tURIs
    // and/or multi-KB headers (for example, BIG cookies).
    //
    // Default buffer size is used if not set.
    ReadBufferSize int

    // Per-connection buffer size for responses' writing.
    //
    // Default buffer size is used if not set.
    WriteBufferSize int

    // Maximum duration for reading the full request (including
body).
    //
    // This also limits the maximum duration for idle keep-alive
    // connections.
    //
    // By default request read timeout is unlimited.
    ReadTimeout time.Duration

    // Maximum duration for writing the full response (including
 body).
    //
    // By default response write timeout is unlimited.
    WriteTimeout time.Duration

    // Maximum number of concurrent client connections allowed p
er IP.
    //
    // By default unlimited number of concurrent connections
    MaxConnsPerIP int

    // Maximum number of requests served per connection.
    //
    // The server closes connection after the last request.
```

```
    // 'Connection: close' header is added to the last response.
    //
    // By default unlimited number of requests may be served per
 connection.
    MaxRequestsPerConn int


    // CheckForUpdates will try to search for newer version of I
ris based on the https://github.com/kataras/iris/releases
    // If a newer version found then the app will ask the he dev
/user if want to update the 'x' version
    // if 'y' is pressed then the updater will try to install th
e latest version
    // the updater, will notify the dev/user that the update is
finished and should restart the App manually.
    // Notes:
    // 1. Experimental feature
    // 2. If setted to true, the app will start the server norma
lly and runs the updater in its own goroutine,
    //    for a sync operation see CheckForUpdatesSync.
    // 3. If you as developer edited the $GOPATH/src/github/kata
ras or any other Iris' Go dependencies at the past
    //    then the update process will fail.
    //
    // Usage: iris.Set(iris.OptionCheckForUpdates(true)) or
    //        iris.Config.CheckForUpdates = true or
    //        app := iris.New(iris.OptionCheckForUpdates(true))
    // Default is false
    CheckForUpdates bool
    // CheckForUpdatesSync checks for updates before server star
ts, it will have a little delay depends on the machine's downloa
d's speed
    // See CheckForUpdates for more
    // Notes:
    // 1. you could use the CheckForUpdatesSync while CheckForUp
dates is false, set this or CheckForUpdates to true not both
    // 2. if both CheckForUpdates and CheckForUpdatesSync are se
tted to true then the updater will run in sync mode, before serv
er server starts.
    //
    // Default is false
```

```
    CheckForUpdatesSync bool

    // DisablePathCorrection corrects and redirects the requeste
d path to the regised path
    // for example, if /home/ path is requested but no handler f
or this Route found,
    // then the Router checks if /home handler exists, if yes,
    // (permant)redirects the client to the correct path /home
    //
    // 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 https:/
/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: Pro
ject%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

    // LoggerOut is the destination for output
    //
    // Default is os.Stdout
    LoggerOut io.Writer
    // LoggerPreffix is the logger's prefix to write at beginnin
```

```
g of each line
    //
    // Default is [IRIS]
    LoggerPreffix string

    // DisableTemplateEngines set to true to disable loading the
 default template engine (html/template) and disallow the use of
 iris.UseEngine
    // default is false
    DisableTemplateEngines bool

    // IsDevelopment iris will act like a developer, for example
    // If true then re-builds the templates on each request
    // default is false
    IsDevelopment bool

    // TimeFormat time format for any kind of datetime parsing
    TimeFormat string

    // 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, thi
s includes any type of render, templates and pure/raw content
    // If you don't want to enable it globaly, you could just us
e the third parameter on context.Render("myfileOrResponse", stru
ctBinding{}, iris.RenderOptions{"gzip": true})
    // defaults to false
    Gzip bool

    // Sessions contains the configs for sessions
    Sessions SessionsConfiguration {
        // Cookie string, the session's client cookie name, for
example: "mysessionid"
        //
        // Defaults to "gosessionid"
        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
 (created_time.Add(Expires)).
        // If you want to delete the cookie when the browser clo
ses, set it to -1 but in this case, the server side's session du
ration is up to GcDuration
        //
        // Defaults to infinitive/unlimited life duration(0)
        Expires time.Duration

        // CookieLength the length of the sessionid's cookie's v
alue, let it to 0 if you don't want to change it
        //
        // Defaults to 32
        CookieLength int

        // GcDuration every how much duration(GcDuration) the me
mory should be clear for unused cookies (GcDuration)
        // for example: time.Duration(2)*time.Hour. it will chec
k every 2 hours if cookie hasn't be used for 2 hours,
        // deletes it from backend memory until the user comes b
ack, then the session continue to work as it was
        //
        // Defaults to 2 hours
        GcDuration time.Duration

        // DisableSubdomainPersistence set it to true in order d
issallow your q subdomains to have access to the session cookie
        //
        // Defaults to false
        DisableSubdomainPersistence bool

        // DisableAutoGC disables the auto-gc running when the s
ession manager is created
        // set to false to disable this behavior, you can call(o
```

```
nnce) manually the GC using the .GC function
        //
        // Defaults to false
        DisableAutoGC bool
    }


    // Websocket contains the configs for Websocket's server int
egration
    Websocket WebsocketConfiguration {
        // WriteTimeout time allowed to write a message to the c
onnection.
        // Default value is 15 * time.Second
        WriteTimeout time.Duration
        // PongTimeout allowed to read the next pong message fro
m the 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 connecti
on
        // Default value is 1024
        MaxMessageSize int64
        // BinaryMessages set it to true in order to denotes bin
ary data messages instead of utf-8 text
        // see https://github.com/kataras/iris/issues/387#issuec
omment-243006022 for more
        // defaults to false
        BinaryMessages bool
        // Endpoint is the path which the websocket server will
listen for clients/connections
        // Default value is empty string, if you don't set it th
e Websocket server is disabled.
        Endpoint string
        // ReadBufferSize is the buffer size for the underline r
eader
        ReadBufferSize int
        // WriteBufferSize is the buffer size for the underline
```

```
    writer
        WriteBufferSize int
    }


    // Tester contains the configs for the test framework, so fa
r we have only one because all test framework's configs are sett
ed by the iris itself
    // You can find example on the https://github.com/kataras/ir
is/glob/master/context_test.go
    Tester TesterConfiguration {
        // ExplicitURL If true then the url (should) be prepende
d manually, useful when want to test subdomains
        // Default is false
        ExplicitURL bool
        // Debug if true then debug messages from the httpexpect
 will be shown when a test runs
        // Default is false
        Debug bool
    }


    // Other are the custom, dynamic options, can be empty
    // this fill used only by you to set any app's options you w
ant
    // for each of an Iris instance
    Other options.Options
}
```

Note that with 2., 3. & 4. you **can serve more than one Iris server** in the same app, when it's necessary.

`.New` **by configuration**

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

//...



myConfig := iris.Configuration{Charset: "UTF-8", IsDevelopment:true, Sessions: iris.SessionsConfiguration{Cookie:"mycookie"}, Websocket: iris.WebsocketConfiguration{Endpoint: "/my_endpoint"}}

iris.New(myConfig)
```

`.New` **by options**

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

//...



iris.New(iris.OptionCharset("UTF-8"), iris.OptionIsDevelopment(true),
iris.OptionSessionsCookie("mycookie"), iris.OptionWebsocketEndpoint("/my_endpoint"))



// if you want to set configuration after the .New use the .Set:

iris.Set(iris.OptionDisableBanner(true))
```

Party

Let's party with Iris web framework!

```go
package main

import "github.com/kataras/iris"

func main() {
    admin := iris.Party("/admin", func(ctx *iris.Context){ ctx.Write("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 for you")
            }

        })
        admin.Get("/", func(c *iris.Context) {
            c.Write("from /admin/ or /admin if you pathcorrection 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: static subdomain and 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

```go
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" is an example on how to catch dynamic/wildcard
 subdomains.
// 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 see an example on how to catch unknown subdomains, dy
namic subdomains,
// like username.mydomain.com:8080.

package main

import "github.com/kataras/iris"

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

```go
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 thi
s 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
```

```go
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 for your routes, you can access them for each request using context's `c.Param("nameoftheparameter")`. Use `c.Params` to get all values as an array ({K,V}).

There's no limit on how long a path can be.

Usage:

```go
package main

import (
    "strconv"

    "github.com/kataras/iris"
)

func main() {
    // Matches /hello/iris,  (if PathCorrection:true match also
/hello/iris/)
    // Doesn't match /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)
    })

    // Matches /profile/iris/friends/1, (if PathCorrection:true
match also /profile/iris/friends/1/)
    // Doesn't match /profile/ or /profile/iris
    // Doesn't match /profile/iris/friends or  /profile/iris/fri
ends
    // Doesn't match /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
            return
        }
        c.HTML(iris.StatusOK, "<b> Hello </b>"+fullname+"<b> wit
h friends ID </b>"+strconv.Itoa(friendID))
    })


    // Route Example:
    // /posts/:id and /posts/new conflict with each other for pe
rformance reasons and simplicity (dynamic value conficts with th
e static 'new').
    // but if you need to have them you can do following:
    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

Named Parameters

```
// Will match any request which's url prefix is "/anything/" and
 has content after that
// Matches /anything/whateverhere/whateveragain or /anything/bla
blabla
// c.Param("randomName") will be /whateverhere/whateveragain, bl
ablabla
// Doesn't match /anything or /anything/ or /something
iris.Get("/anything/*randomName", func(c *iris.Context) { } )
```

Static files

Serve a static directory

```
// StaticHandler returns a HandlerFunc to serve static system di
rectory
// Accepts 5 parameters
//
// first param 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
```

```go
// * 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 the second parameter is empty, otherwise the requestPath i
s the second parameter
// it uses gzip compression (compression on each request, no fil
e cache)
StaticServe(systemPath string, requestPath ...string)
```

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

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

```go
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:

```go
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:

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

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

```go
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")
}
```

Disabling caching

`Static` , `StaticFS` and `StaticWeb` functions automatically cache the given files for a period of time (default 20 seconds). In certain situations you don't want that caching to happen (development etc.).

Caching can be disabled by setting `github.com/kataras/iris/config` 's `StaticCacheDuration` to `time.Duration(1)` **before calling any of the named functions**. Setting `StaticCacheDuration` to `time.Duration(0)` will reset the cache time to 10 seconds (as specified in fasthttp).

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` .

```go
// ./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 this function call
// for example: ctx.Response.Header.Set("Content-Type","thecontent/type")
SendFile(filename string, destinationName string)
```

```go
package main

import "github.com/kataras/iris"

func main() {

    iris.Get("/servezip", func(c *iris.Context) {
        file := "./files/first.zip"
        c.SendFile(file, "saveAsName.zip")
    })

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

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

Send files

```go
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

```go
// Config keeps the options for the 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 part 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
}
```

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

Installation

```sh
sh go get -u github.com/kataras/go-mailer
```

Example

File: `./main.go`

```go
package main

import (
    "github.com/kataras/go-mailer"
    "github.com/kataras/iris"
)

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

    cfg := mail.Config{
        Host:     "smtp.mailgun.org",
        Username: "postmaster@sandbox661c307650f04e909150b37c0f3b2f09.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...)
```

```go
            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 renderingt a 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 is an 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`

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


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

Render

Think of 'Render' as an action which sends/responds with rich content to the client.

The render actions are separated into two categories:

- **Responses** send content using `Serialize Engines` which use the `Content-Type` as a `Key (explained later)`. (i.e. JSON, XML etc.)

- **Templates** send content using `Template Engines` which use file name extensions. (i.e. Markdown, Jade etc.)

Serialize 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-in template engines, you can create your own and 'inject' it to the iris station. You can also use more than one template engines at the same time (when the file extensions are different from each other).

- examples are located here.

Install

Default Serializers* are already installed when Iris has been installed.

Iris' Station configuration

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

```
iris.Config.Gzip = true // compresses/gzips response content to
the client (same for Template Engines), defaults to false
iris.Config.Charset = "UTF-8" // defaults to "UTF-8" (same for T
emplate Engines also)

// or
iris.Set(iris.OptionGzip(true), iris.OptionCharset("UTF-8"))
// or
iris.New(iris.OptionGzip(true), iris.OptionCharset("UTF-8"))
// or
iris.New(iris.Configuration{ Gzip:true, Charset: "UTF-8" })
```

They can be overriden for specific `Render` actions:

```
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 scared about the 'big' article, a serialize engine(serializer, old: Serializer) is very simple and is easy to understand. Let's see what built-in response types are available in `iris.Context`.

```
package main
```

```go
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")
}
```

Text Serializer

```go
package main

import "github.com/kataras/iris"

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

    myString := "this is just a simple string which you can already 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", myString)
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.Render("text/plain", myString, iris.RenderOptions{"charset": "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 Serializer

You can create a custom Serializer using a func or an interface which implements the `serializer.Serializer` which contains a simple function: `Serialize(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 Serializers, custom or not: say for example, that you want a static'footer/suffix' on your content.

If a Serializer 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 the `text/plain` content type, because you can see its results easly.

```
// You can create a custom serialize engine(serializer) using a
func or an interface which implements the
// serializer.Serializer which contains a simple function: Seria
lize(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

// Let's do a 'trick' here, which works for all other serialize
engine(serializer)s, custom or not:

// say for example, that you want a static'footer/suffix' on you
r content, without the need to create & register a middleware fo
r that, per route or globally
// you want to be even more organised.
//
// IF a serialize engine(serializer) has the same key and the sa
me content type then the contents are appended and the final res
ult will be rendered to the client.

// Enough with my 'bad' english, let's code something small:
```

```go
package main

import (
    "github.com/kataras/go-serializer"
    "github.com/kataras/go-serializer/text"
    "github.com/kataras/iris"
)

// 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.
func main() {
    // we are registering the default text/plain,  and after we will register the 'appender' only
    // we have to register the default because we will add more serialize engine(serializer)s with the same content,
    // iris will not register this by-default if other serialize engine(serializer) with the corresponding ContentType already exists
    iris.UseSerializer(text.ContentType, text.New())

    // register a serialize engine(serializer) serializer.Serializer
    iris.UseSerializer(text.ContentType, &CustomTextEngine{})
    // register a serialize engine(serializer) with func
    iris.UseSerializer(text.ContentType, serializer.SerializeFunc(func(val interface{}, options ...map[string]interface{}) ([]byte, error) {
        return []byte("\nThis is the static SECOND AND LAST suffix!"), nil
    }))

    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 serialiser.Serializer
 implementation:

// CustomTextEngine the serialize engine(serializer) which appen
ds a simple string on the default's text engine
type CustomTextEngine struct{}

// Implement the serializer.Serializer
func (e *CustomTextEngine) Serialize(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.SerializeToString

SerializeToString gives you the result of the Serializer'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() {

    // SerializeToString gives you the result of the serialize e
ngine(serializer)'s work, it doesn't renders to the client but y
ou can use
    // this function to collect the end result and send it via e
-mail to the user, or anything you can imagine.

    // Note that: iris.SerializeToString is called outside of th
```

```
e context, using your iris $instance (iris. is the default)

    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 serialize engine(serializer)S, bu
t I am not doing the same example for all engines again :) (the
same you can do with templates using the iris.TemplateString)
          htmlContents := iris.SerializeToString("text/markdown",
markdownContents, iris.RenderOptions{"charset": "8859-1"}) // de
fault 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 Serializers

JSON Serializer

```go
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.
```

```go
    })

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

```go
package main

import (
    "github.com/kataras/go-serializer/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 ser
ialize engine(serializer) in order to change its internal configs

    // this is one of the reasons you need to import a default e
ngine,(template engine or serialize engine(serializer))
    /*
        type Config struct {
            Indent        bool
            UnEscapeHTML  bool
            Prefix        []byte
            StreamingJSON bool
        }
    */
    iris.UseSerializer(json.ContentType, json.New(json.Config{
        Prefix: []byte("MYPREFIX"),
    })) // you can use anything as the second parameter, the jso
n.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"}, iri
s.RenderOptions{"charset": "8859-1"})
    }


    //second example,
    // imagine that we need the context.JSON to be listening to
our "application/json" serialize engine(serializer) with a custo
m prefix (we did that before)
    // but we also want a different renderer, but again applicat
ion/json content type, with Indent option setted to true:
    iris.UseSerializer("json2", json.New(json.Config{Indent: true
}))
    json2Handler := func(ctx *iris.Context) {
        ctx.Render("json2", myjson{Name: "My iris"})
        ctx.SetContentType("application/json")
    }

    iris.Get("/", jsonHandlerSimple)

    iris.Get("/render", jsonHandlerWithRender)

    iris.Get("/json2", json2Handler)

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

JSONP Serializer

```
package main
```

```go
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")
}
```

```go
package main

import (
    "github.com/kataras/go-serializer/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 serialize engine(serializer))
    /*
        type Config struct {
            Indent   bool
            Callback string // the callback can be override by t
he context's options or parameter on context.JSONP
        }
    */
    iris.UseSerializer(jsonp.ContentType, jsonp.New(jsonp.Config
{
        Indent: true,
    }))
    // 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"})
    }
```

```go
    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.UseSerializer("jsonp2", jsonp.New(jsonp.Config{Callback
: "callbackName"}))
    // yes the UseSerializer returns a function which you can ma
p the content type if it's not declared on the key
    handlerJsonp2 := func(ctx *iris.Context) {
        ctx.Render("jsonp2", myjson{Name: "My iris"})
        ctx.SetContentType("application/javascript")
    }

    iris.Get("/", handlerSimple)

    iris.Get("/render", handlerWithRender)

    iris.Get("/jsonp2", handlerJsonp2)

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

XML Serializer

```go
package main

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

```go
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"})
    })
```

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

```go
package main

import (
    encodingXML "encoding/xml"

    "github.com/kataras/go-serializer/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 serialize engine(serializer))
    /*
        type Config struct {
            Indent bool
            Prefix []byte
        }
    */
    iris.UseSerializer(xml.ContentType, xml.New(xml.Config{
        Indent: true,
    }))
    // 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.
```

```go
    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/xml" as content type, with prefix option setted by configuration:
    iris.UseSerializer("xml2", xml.New(xml.Config{Prefix: []byte("")})) // if you really use a PREFIX it will be not valid xml, use it only for special cases
    handlerXML2 := func(ctx *iris.Context) {
        ctx.Render("xml2", myxml{First: "first attr", Second: "second attr"})
        ctx.SetContentType("text/xml; charset=" + iris.Config.Charset)
    }

    iris.Get("/", handlerSimple)

    iris.Get("/render", handlerWithRender)

    iris.Get("/xml2", handlerXML2)

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

Markdown Serializer

```go
package main

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

```go
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 with
     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 let 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 content, see
     [this example](https://github.com/russross/blackfriday#sanit
```

```
ize-untrusted-content).

*   **Fast processing**. It is fast enough to render on-demand in
    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 global
    shared state.

*   **Minimal dependencies**. Blackfriday only depends on standard
    library packages in Go. The source code is pretty
    self-contained, so it is easy to add to any project, including
    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 serialize engine(serializer) 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)
    })
```

```go
    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")
}
```

```go
package main

import (
    "github.com/kataras/go-serializer/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 with
    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 let 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 content, see
    [this example](https://github.com/russross/blackfriday#sanitize-untrusted-content).

*   **Fast processing**. It is fast enough to render on-demand in
    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 global
    shared state.

*   **Minimal dependencies**. Blackfriday only depends on standard
    library packages in Go. The source code is pretty
    self-contained, so it is easy to add to any project, including
```

```
    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 serialize engine(serializer))
    /*
        type Config struct {
            MarkdownSanitize bool
        }
    */
    iris.UseSerializer(markdown.ContentType, markdown.New())
    // you can use anything as the second parameter,
    // the markdown.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/ht
ml" as 'content type' (this is the real content type we want to
render with, at the first ctx.Render the text/markdown key is co
nverted automatically to text/html without need to call SetConte
ntType), with MarkdownSanitize option setted to true:
    iris.UseSerializer("markdown2", markdown.New(markdown.Config
{MarkdownSanitize: true}))
    handlerMarkdown2 := func(ctx *iris.Context) {
        ctx.Render("markdown2", markdownContents, iris.RenderOpt
ions{"gzip": true})
```

```
        ctx.SetContentType("text/html")
    }

    iris.Get("/", handlerWithRender)

    iris.Get("/markdown2", handlerMarkdown2)

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

Data(Binary) Serializer

```go
package main

import "github.com/kataras/iris"

func main() {
    myData := []byte("some binary data or a program here which will 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-stream", myData)
    })

    iris.Get("/alternative_3", func(ctx *iris.Context) {
        ctx.Render("application/octet-stream", myData, iris.RenderOptions{"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")
}
```

Serialize Engines

- examples are located here.

- You can contribute to Serializers, click here to navigate to the reository.

Install

Install the go-template package.

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

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 each request, defaults to false
iris.Config.Gzip  = true // compressed gzip contents to the client, the same for Serializers also, defaults to false
iris.Config.Charset = "UTF-8" // defaults to "UTF-8", the same for Serializers also

// or
iris.Set(iris.OptionIsDevelopment(true),iris.OptionGzip(true), iris.OptionCharset("UTF-8"))
// or
iris.New(iris.OptionIsDevelopment(true),iris.OptionGzip(true), iris.OptionCharset("UTF-8"))
// or
iris.New(iris.Configuration{IsDevelopment:true, Gzip:true, Charset: "UTF-8" })
```

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

```
func(ctx *iris.Context){
    ctx.Render("templateFile.html", anyBindingStruct{}, iris.RenderOptions{"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, defaulted

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

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

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

import "github.com/kataras/iris"

// nothing to do, defaults to ./templates and .html extension, no 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 templates 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"})
}
```

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

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

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

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

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

import (
    "github.com/kataras/go-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")
}
```

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

```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>
```

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

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

    <h1>Page 1</h1>

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

</div>
```

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

import (
    "github.com/kataras/go-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>
```

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

    <h1>Page 1</h1>

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

</div>
```

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

import (
    "github.com/kataras/go-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")
}
```

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

```go
// ./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 kataras/go-template/html because
iris uses this as the default engine if no other template engine
 has been registered.
package main

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

```go
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())


}
```

```html
<!-- ./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 also 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-
```

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

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

func main() {

    wildcard := iris.Party("*.")
    {
        wildcard.Get("/mypath", emptyHandler)("dynamic-subdomain1")
        wildcard.Get("/mypath2/:param1/:param2", emptyHandler)("dynamic-subdomain2")
        wildcard.Get("/mypath3/:param1/statichere/:param2", emptyHandler)("dynamic-subdomain3")
        wildcard.Get("/mypath4/:param1/statichere/:param2/:other
```

```go
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

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

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

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

func main() {

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

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

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

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

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

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

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

```go
    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")
}
```

```go
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

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

```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>
```

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

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

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

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

</div>
```

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

import (
```

```go
    "github.com/aymerick/raymond"
    "github.com/kataras/go-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"}}
```

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

import (
    "html/template"

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

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

```go
}

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")
}
```

```jade
<!-- ./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
```

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

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

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

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

```go
        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
    })
```

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


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


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

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

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

type mypage struct {
    Name string
}

func main() {

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

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("basic.amber", mypage{"iris"}, iris.RenderOptions{"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 }}` .

```go
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.

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

Template Engines

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

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

func main() {

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

    iris.Get("/", func(ctx *iris.Context) {
        // THIS WORKS WITH ALL TEMPLATE ENGINES, but I am not doing the same example for all engines again :) (the same you can do with templates using the iris.SerializeToString)
        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 repository.

Gzip

Gzip compression is easy.

Activate **auto-gzip** for all responses and template engines, just set `iris.Config.Gzip = true` or `iris.New(iris.OptionGzip(true))` or `iris.Set(OptionGzip(true))` . You can also enable gzipping for specific `Render()` calls:

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

```
// WriteGzip writes the gzipped body of the response to w.
//
// Gzips the response body and sets the 'Content-Encoding: gzip'
// header before writing response to w.
//
// WriteGzip doesn't flush the response to w for performance rea
sons.
WriteGzip(w *bufio.Writer) error


// WriteGzip writes the gzipped body of the response to w.
//
// Level is the desired compression level:
//
//      * CompressNoCompression
//      * CompressBestSpeed
//      * CompressBestCompression
//      * CompressDefaultCompression
//
// Gzips the response body and sets the 'Content-Encoding: gzip'
// header before writing response to w.
//
// WriteGzipLevel doesn't flush the response to w for performanc
e reasons.
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.

```go
// StreamWriter registers the given stream writer for populating
 the 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

```go
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 client.
        if err := w.Flush(); err != nil {
            return
        }
        time.Sleep(time.Second)
    }
}
```

To achieve the oposite make use of the `StreamReader` :

```
// StreamReader sets the response body stream and optionally body size.
//
// If bodySize is >= 0, then the bodyStream must provide the exact 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!

```go
// 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 the 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
tp) has a strange bug, which I cannot solve at the moment.
// This is the reason why this function exists and should be use
d instead of fasthttp's built in function.
VisitAllCookies(visitor func(key string, value string))
```

How to use:

```go
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 the 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.

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

// GetFlash gets 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 error is filled.
// after the request's life the value is removed
GetFlash(key string) (value string, err error)

// GetFlashes returns all the flash messages which are available for this request
GetFlashes() map[string]string
```

Flash messages

Example

```go
package main

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

func main() {

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

    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 sets 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

```go
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

```go
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

Example form

```go
//./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")
}
```

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

In form html

- Use symbol `.` to access a field/key of a structure or map. (i.e., `struct.key`)
- Use `[int_here]` to access an 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 an input or form starts lowercase.

```go
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"`
```

```go
    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 Decoder per Object

`BodyDecoder` gives the ability to set a custom decoder **per passed object** when `context.ReadJSON` and `context.ReadXML`

```go
// BodyDecoder is an interface which any struct can implement in
 order to customize the decode action
// from ReadJSON and ReadXML
//
// Trivial example of this could be:
// type User struct { Username string }
//
// func (u *User) Decode(data []byte) error {
//      return json.Unmarshal(data, u)
// }
//
// the 'context.ReadJSON/ReadXML(&User{})' will call the User's
// Decode option to decode the request body
//
// Note: This is totally optionally, the default decoders
// for ReadJSON is the encoding/json and for ReadXML is the enco
ding/xml
type BodyDecoder interface {
    Decode(data []byte) error
}
```

> for a usage example go to
> https://github.com/kataras/iris/blob/master/context_test.go#L262

Custom HTTP Errors

You can define your own handlers when http error occurs.

```go
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")
        // or ctx.Render, ctx.HTML any render method you want
        ctx.Log("http status: 500 happened!")
    })

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

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

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

    iris.Listen(":80")

}
```

Context

The context source code can be found here. Keep note that many context's functions are not written here, use IDE/Editors with `auto-complete` feature.

```
*fasthttp.RequestCtx
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
```

```go
        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(config ...Config) iris.HandlerFunc
```

How to use

```go
package main

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

/*
With configs:

errorLogger := logger.New(logger.Config{
    // 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())
*/
```

```go
func main() {

    iris.Use(logger.New())

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

    iris.Listen(":8080")

}
```

HTTP access control

[This is a middleware](#).

Some security work for between you and 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:

```go
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

```go
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

```go
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 set it it's "Authorization Required"
        ContextKey: "mycustomkey",            // if you don't set 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.Context) {
            username := ctx.GetString("mycustomkey") //  the Contextkey from the authConfig
            ctx.Write("Hello authenticated user: %s ", username)
        })
    */

    // to party

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

```go
        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 enables you to connect your clients using famous websites login APIs, it is a bridge to the goth library.

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"
```

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

// register your auth via configs, providers with non-empty
// values 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
}


// 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 ha
ndler)
    // if the user succeed to logged in
    // client comes here from: localhost:3000/config.RouteName/l
owercase_provider_name/callback 's first handler,
    // but the previous url is the localhost:3000/config.RouteNa
me/lowercase_provider_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:

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

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

```html
<!-- ./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 iris-contrib/gothic which is like the original goth but converted to work with Iris.

Example:

```go
package main

import (
    "html/template"
    "os"

    "sort"

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

```go
    "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 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 authorize in Twitter provider, use this instead.
        // twitter.NewAuthenticate(os.Getenv("TWITTER_KEY"), os.Getenv("TWITTER_SECRET"), "http://localhost:3000/auth/twitter/callback"),
```

```go
        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
```

```go
        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"
```

```go
    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, ProvidersMap: 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)
```

```go
        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

```go
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) {
```

```go
            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").(*
jwt.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

```go
package main

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

func main() {
    s := secure.New(secure.Options{
        // AllowedHosts is a list of fully qualified domain names

        // that are allowed. Default is empty list,
        // which allows any and all host names.
        AllowedHosts:            []string{"ssl.example.com"},


        // If SSLRedirect is set to true, then only allow HTTPS
requests.
        // Default is false.
        SSLRedirect:             true,
        // If SSLTemporaryRedirect is true,
        // then a 302 will be used while redirecting.
        // Default is false (301).
        SSLTemporaryRedirect:    false,
        // SSLHost is the host name that is used to
```

```
        // redirect HTTP requests to HTTPS.
        // Default is "", which indicates to use the same host.
        SSLHost:                  "ssl.example.com",
        // SSLProxyHeaders is set of header keys with associated
 values
        // that would indicate a valid HTTPS request.
        // Useful when using Nginx: `map[string]string{"X-Forwar
ded-Proto": "https"}`.
        // Default is blank map.
        SSLProxyHeaders:         map[string]string{"X-Forwarded-
Proto": "https"},
        // 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 hea
der. Default is false.
        STSSeconds:              315360000,


        // If STSPreload is set to true, the `preload`
        // flag will be appended to the Strict-Transport-Securit
y header.
        // Default is false.
        STSPreload:              true,
        // STS header is only included when the connection is HT
TPS.
        // If you want to force it to always be added, set to tr
ue.
        // `IsDevelopment` still overrides this. Default is fals
e.
        ForceSTSHeader:          false,
        // If FrameDeny is set to true, adds the X-Frame-Options
 header with
        // the value of `DENY`. Default is false.
        FrameDeny:               true,
```

```
        // CustomFrameOptionsValue allows the X-Frame-Options he
ader
        // value to be set with a custom value.
        // This overrides the FrameDeny option.
        CustomFrameOptionsValue: "SAMEORIGIN",
        // If ContentTypeNosniff is true, adds the X-Content-Typ
e-Options
        // header with the value `nosniff`. Default is false.
        ContentTypeNosniff:      true,
        // If BrowserXssFilter is true, adds the X-XSS-Protectio
n header
        // with the value `1;mode=block`. Default is false.
        BrowserXSSFilter:        true,
        // ContentSecurityPolicy allows the Content-Security-Pol
icy
        // header value to be set with a custom value. Default i
s "".
        ContentSecurityPolicy:   "default-src 'self'",
        // PublicKey implements HPKP to prevent
        // MITM attacks with forged certificates. Default is "".
        PublicKey:               `pin-sha256="base64+primary==";
 pin-sha256="base64+backup=="; max-age=5184000; includeSubdomain
s; report-uri="https://www.example.com/hpkp-report"`,
        // This will cause the AllowedHosts, SSLRedirect,
        //..and STSSeconds/STSIncludeSubdomains options to be ig
nored during development.
        // When deploying to production, be sure to set this to
false.
        IsDevelopment: true,
    })

    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")
}
```

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

Sessions

If you notice a bug or issue post it here.

- Cleans the temp memory when a session is idle, and re-allocates it to the temp memory when it's necessary. The most used sessions are optimized to be in the front of the memory's list.

- Supports any type of database, currently only Redis and LevelDB.

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 application.

Instead of storing large and constantly changing data via cookies in the user's browser (i.e. CookieStore), **only a unique identifier is stored on the client side** called a "session id". This session id is passed to the web server on every request. The web application uses the session id as the key for retrieving the stored data from the database/memory. The session data is then available inside the iris.Context.

Here you see two different ways to use the sessions, we are using the first in this example. There are no performance differences.

How to use

```go
package main

import    "github.com/kataras/iris"

func main() {

    // These are the optional fields to configurate sessions,
    // using the station's Config field (iris.Config.Sessions)

    // Cookie string, the session's client cookie name, for exam
```

```
ple: "qsessionid"
    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)).
    // If you want to delete the cookie when the browser closes,
 set it to -1 but in this case, the server side's session durati
on is up to GcDuration
    //
    // Default infinitive/unlimited life duration(0)
    Expires time.Duration

    // CookieLength the length of the sessionid's cookie's value
, let it to 0 if you don't want to change it
    // Defaults to 32
    CookieLength int

    // 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 q 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")
    })
```

```go
    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 a string.
        // returns an empty string if the key was not found.
        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 whatever
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 whateve
r 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 is located here.

```go
package main

import (
    "github.com/kataras/go-sessions/sessiondb/redis"
    "github.com/kataras/go-sessions/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 set here
```

```go
        c.Write("All ok session set to: %s", c.Session().GetStri
ng("name"))
    })

    iris.Get("/get", func(c *iris.Context) {
        // get a specific key as a string.
        // returns an empty string if the key was not found.
        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 real-time data transfer from and to the server**.

Read more about Websockets on Wikipedia.

Configuration

```
type WebsocketConfiguration 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
    // BinaryMessages set it to true in order to denotes binary
```

```
data messages instead of utf-8 text
    // see https://github.com/kataras/iris/issues/387#issuecomme
nt-243006022 for more
    // defaults to false
    BinaryMessages bool
    // 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
    // ReadBufferSize is the buffer size for the underline reader


    ReadBufferSize int
    // WriteBufferSize is the buffer size for the underline writ
er
    WriteBufferSize int
    // Headers  if true then the client's headers are copy to th
e websocket connection
    //
    // Default is true
    Headers bool
    // Error specifies the function for generating HTTP error re
sponses.
    //
    // The default behavior is to store the reason in the contex
t (ctx.Set(reason)) and fire any custom error (ctx.EmitError(sta
tus))
    Error func(ctx *Context, status int, reason string)
    // CheckOrigin returns true if the request Origin header is
acceptable. If
    // CheckOrigin is nil, the host in the Origin header must no
t be set or
    // must match the host of the request.
    //
    // The default behavior is to allow all origins
    // you can change this behavior by setting the iris.Config.W
ebsocket.CheckOrigin = iris.WebsocketCheckSameOrigin
    CheckOrigin func(ctx *Context) bool
}
```

```
iris.Config.Websocket.Endpoint = "/myEndpoint"
// or
iris.Set(iris.OptionWebsocketEndpoint("/myEndpoint"))
// or
iris.New(iris.Configuration{Websocket: iris.WebsocketConfigurati
on{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(iris.All).Emit/EmitMessage...

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

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


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

// Force-disconnect the client from the server-side
Disconnect() error
```

How to use

Server-side

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

import (
    "fmt"

    "github.com/kataras/iris"
)

type clientPage struct {
    Title string
    Host  string
}


func main() {
```

```go
    iris.Static("/js", "./static/js", 1)

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

    // the path at which the websocket client should register it
self to
    iris.Config.Websocket.Endpoint = "/my_endpoint"
    // for Allow origin you can make use of the middleware
    //iris.Config.Websocket.Headers["Access-Control-Allow-Origin
"] = "*"

    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(iris.Broadcast).Emit("chat", "Message from: "
+c.ID()+"-> "+message)

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

            // send the message to the whole room,
            // all connections which are inside this room will r
eceive 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

```javascript
// 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></center></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.

```go
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](#).

Safely recover the server from a panic.

```go
package main

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

func main() {
    //iris.Use(recovery.New(os.Stdout))
    // this is an optional parameter, you can skip it, the default is os.Stderr
    iris.Use(recovery.New())
    i := 0
    iris.Get("/", func(ctx *iris.Context) {
        i++
        if i%2 == 0 {
            panic("a panic here")
            return

        }

        ctx.Next()

    }, func(ctx *iris.Context) {
        ctx.Write("Hello, refresh one time more to get panic!")
    })

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

Plugins

Plugins are modules which get injected into Iris' flow. They're like middleware for the Iris framework itself. Middlewares start their actions after the server processes requests and get executed on every request, plugins on the other hand start working when you registered them. Plugins work with the framework's code, they have access to the *iris.Framework, so they are able register routes, start a second server, read Iris' configs or edit them and so on. The Plugin interface looks this:

```go
type (
    // Plugin just an empty base for plugins
    // A Plugin can be added with: .Add(PreListenFunc(func(*Framework))) and so on... or
    // .Add(myPlugin{},myPlugin2{}) 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 return the name of the plugin, a name is unique.
        // name has to be not dependent from other methods of the plugin,
        // because it is being called even before Activate()
        GetName() string
    }

    // pluginGetDescription implements the GetDescription() string method
    pluginGetDescription interface {
        // GetDescription has to return the description of what the plugins is used for
        GetDescription() string
    }
```

```go
    // 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's called only once
        //
        // 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 is called only once, BEFORE the server is s
tarted (if .Listen called)
        //  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 is called once, AFTER the server is starte
d (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 is called once, 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)

)
```

```go
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 regested 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) {
```

```go
    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)
*/
```

An example of one plugin which is under development is Iris control, a web interface that gives you remote control to your Iris web server. You can find it's code here.

Take a look at the plugin.go, it's easy to make your own plugin.

Custom callbacks can be maden with third-party package go-events.

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
```

```go
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 langu
age 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 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

```go
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 options just: typescript.New()

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

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

Enable web browser editor

```go
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](#) language.

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

```go
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",
Port: 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 readable to anyone who could intercept the HTTP request (if TLS is not used). Read more here.

> The editor can't work if the directory doesn't contain 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 still work in progress.

It gives you access to information/stats about your iris server via a web interface.

> You need an internet connection the first time you will run this plugin, because the assets don't exist in the 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

```go
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")
}
```