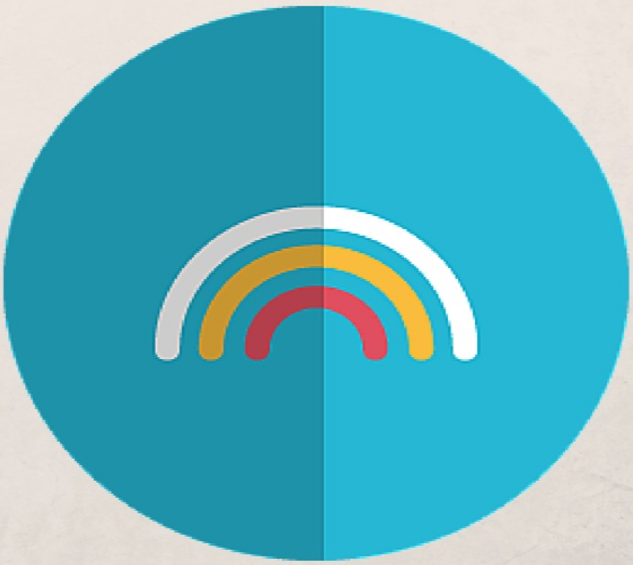
Iris web framework

The complete guide



brought to you absolutely free

github.com/kataras/iris

Table of Contents

Introduction 1.1
Features 1.2
Versioning 1.3
Install 1.4
Hi 1.5
Transport Layer Security 1.6
Handlers 1.7
- Using Handlers 1.7.1
- Using HandlerFuncs 1.7.2
- Using custom struct for a complete API 1.7.3
- Using native http.Handler 1.7.4
  - Using native http.Handler via iris.ToHandlerFunc() 1.7.4.1

Middleware 1.8
API 1.9
Declaration 1.10
Configuration 1.11
Party 1.12
Subdomains 1.13
Named Parameters 1.14
Static files 1.15
Send files 1.16
Send e-mails 1.17
Render 1.18
- REST 1.18.1
- Templates 1.18.2

Gzip 1.19
Streaming 1.20

Cookies	1.21
Flash messages	1.22
Body binder	1.23
Custom HTTP Errors	1.24
Context	1.25
Logger	1.26
HTTP access control	1.27
Basic Authentication	1.28
OAuth, OAuth2	1.29
JSON Web Tokens(JWT)	1.30
Secure	1.31
Sessions	1.32
Websockets	1.33
Graceful	1.34
Recovery	1.35
Plugins	1.36
Internationalization and Localization	1.37
Easy Typescript	1.38
Browser based Editor	1.39
Control panel	1.40
https://github.com/iris-contrib/examples	1.41

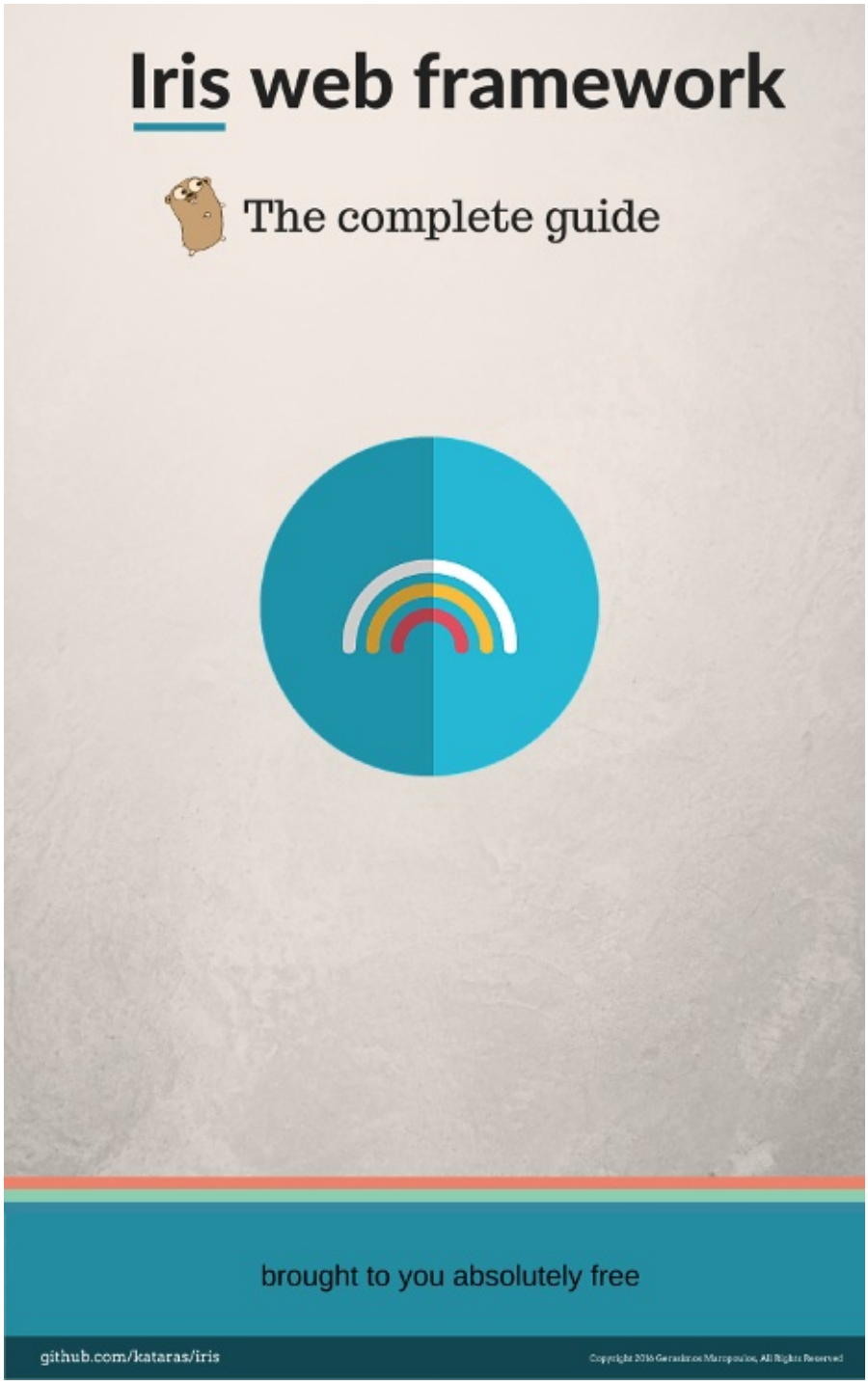


Table of Contents

- Introduction
- Features
- Versioning
- Install
- Hi
- Transport Layer Security

- Handlers
  - Using Handlers
  - Using HandlerFuncs
  - Using custom struct for a complete API
  - Using native http.Handler
    - Using native http.Handler via iris.ToHandlerFunc()
- Middleware
- API
- Declaration
- Configuration
- Party
- Subdomains
- Named Parameters
- Static files
- Send files
- Send e-mails
- Render
  - REST
  - Templates
- Gzip
- Streaming
- Cookies
- Flash messages
- Body binder
- Custom HTTP Errors
- Context
- Logger
- HTTP access control
- Basic Authentication
- OAuth, OAuth2
- JSON Web Tokens(JWT)
- Secure
- Sessions
- Websockets
- Graceful
- Recovery

- Plugins
- Internationalization and Localization
- Easy Typescript
- Browser based Editor
- Control panel
- Examples here: https://github.com/iris-contrib/examples

Why

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

When you think about building web applications and web APIs, or simply building HTTP servers in Go, does your mind go to the standard net/http package? Then you have to deal with some common situations like dynamic routing (a.k.a parameterized), security and authentication, real-time communication and many other issues that net/http doesn't solve.

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

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

The truth

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

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

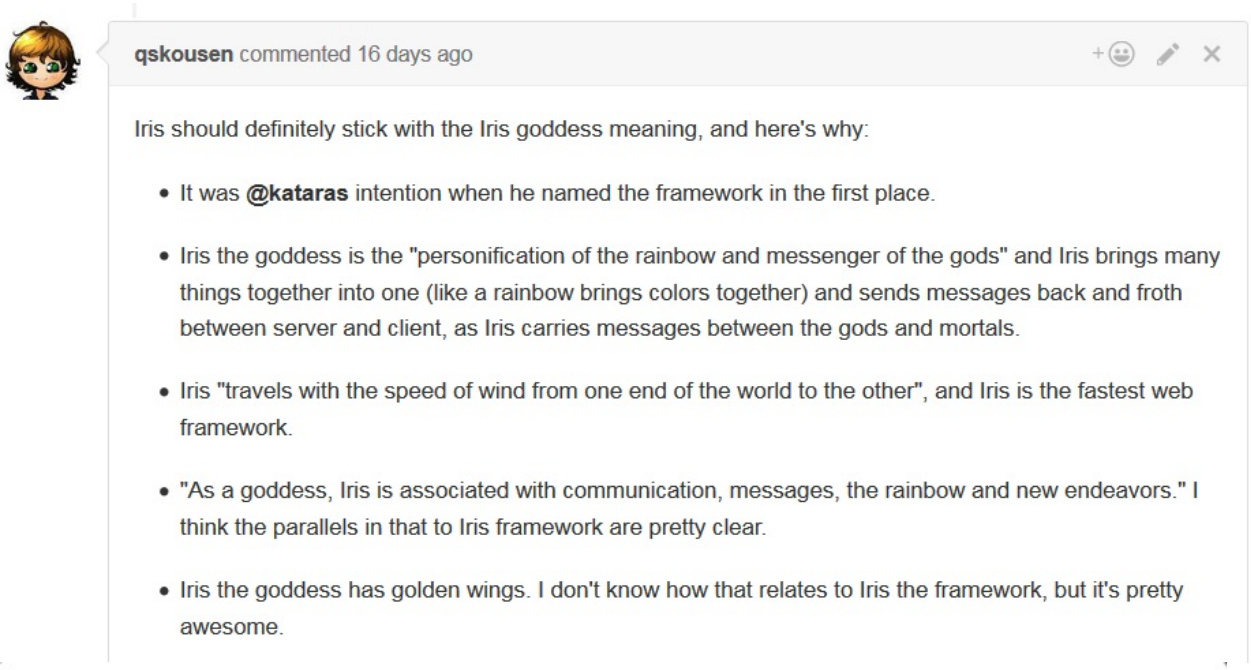
> **'Makis, don't lose hope, give at least a chance to Golang. Try to build something totally new without basing it off the "slow" code you saw earlier; learn the secrets of this language and make *others* follow your steps!'**.

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

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

Two months later, I'm writing this intro.

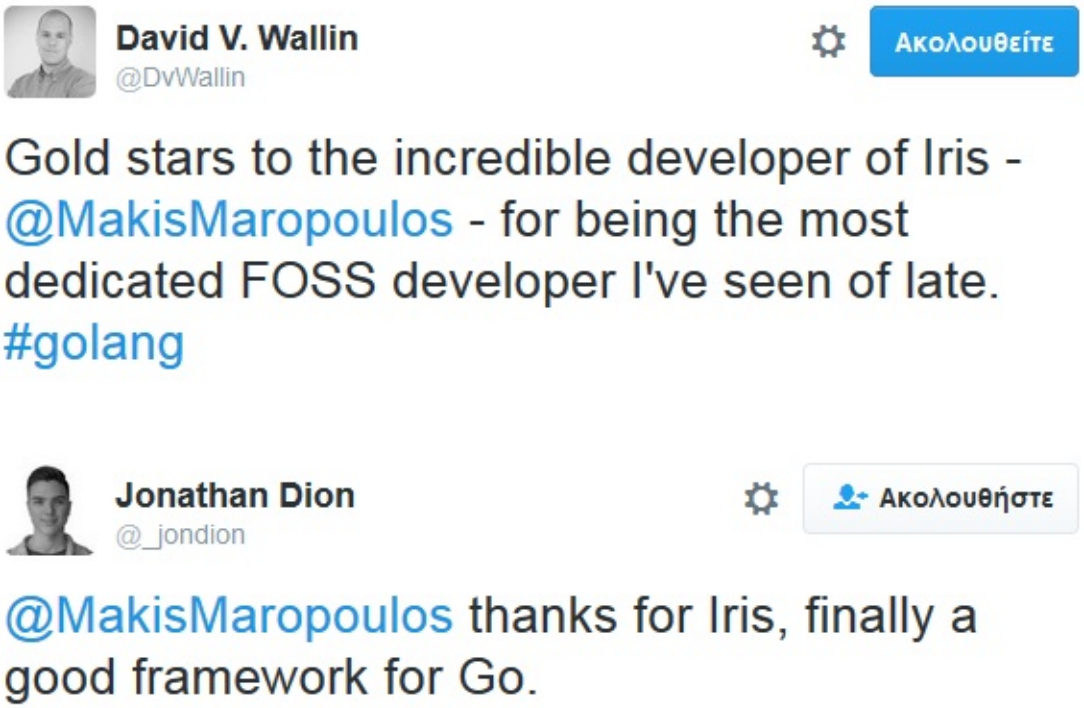
I'm still here because Iris has succeed in being the fastest go web framework



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.

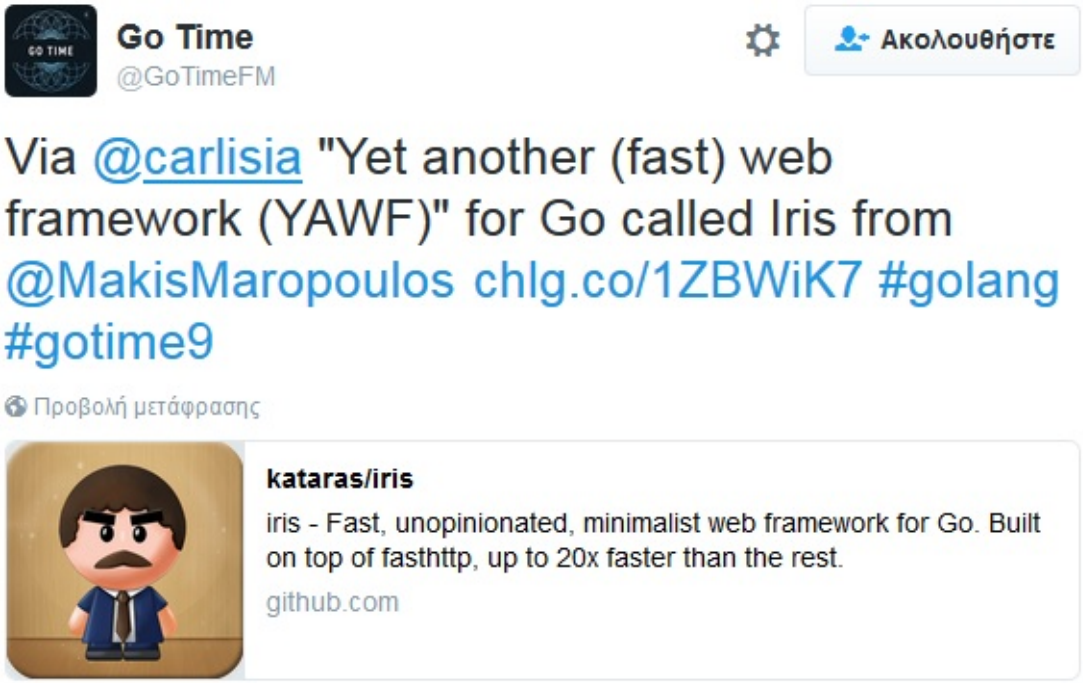


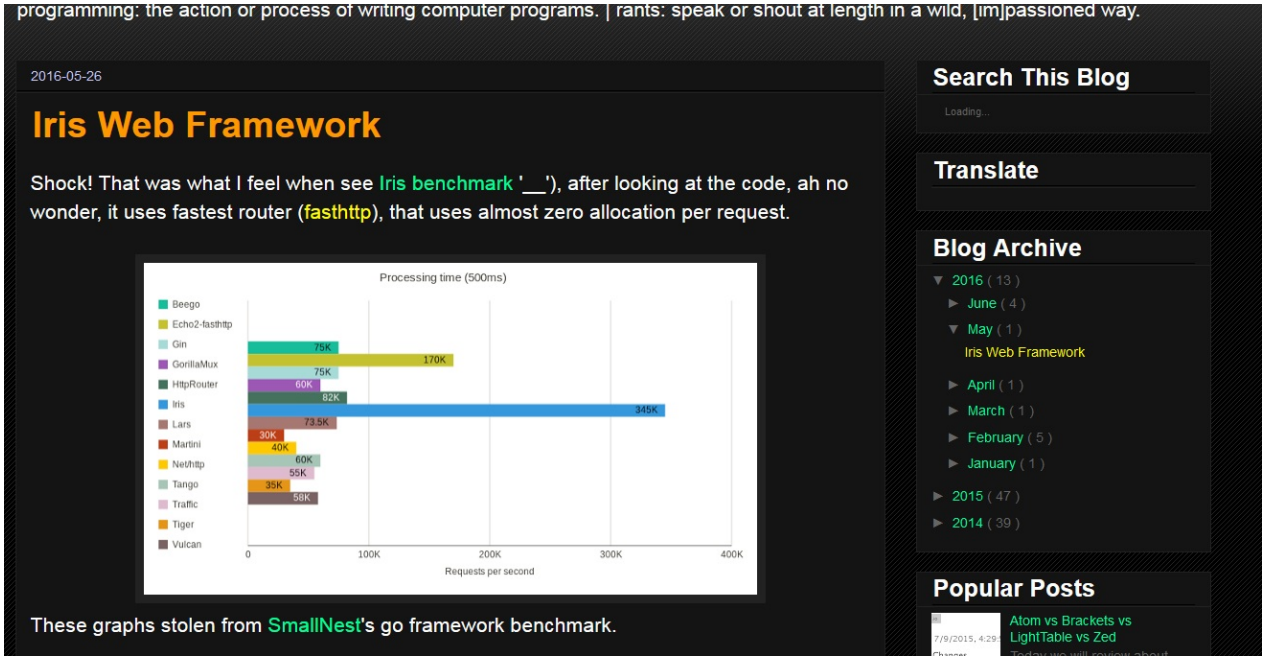
David V. Wallin
@DvWallin
Ακολουθείτε

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.

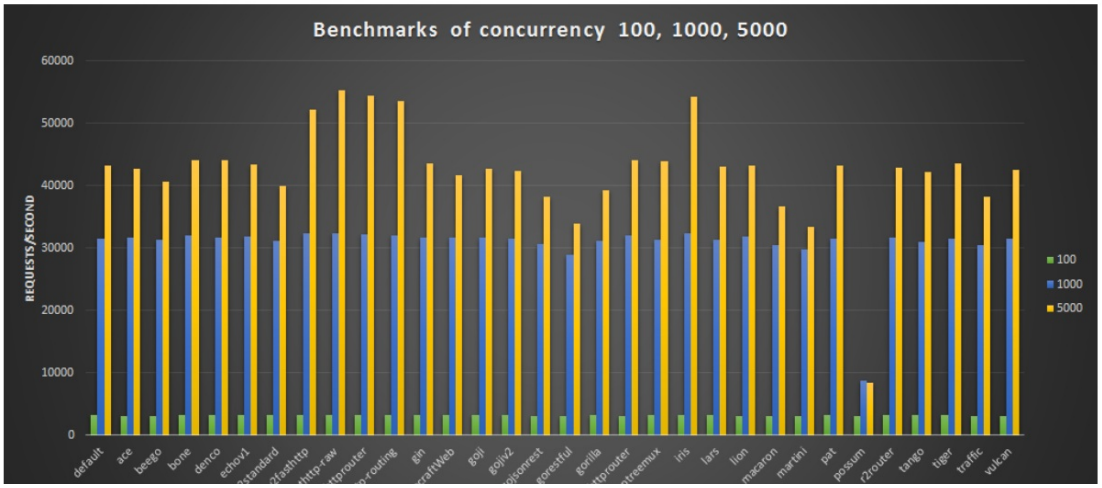

Go Time
@GoTimeFM
Ακολουθήστε
Via @carlisia "Yet another (fast) web framework (YAWF)" for Go called Iris from @MakisMaropoulos chlg.co/1ZBWiK7 #golang #gotime9
Προβολή μετάφρασης
kataras/iris
iris - Fast, unopinionated, minimalist web framework for Go. Built on top of fasthttp, up to 20x faster than 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)
Beego
Echo2-fasthttp
Gin
GorillaMux
HttpRouter
Iris
Lars
Martini
Net/http
Tango
Traffic
Tiger
Vulcan
75K
170K
75K
60K
82K
345K
73.5K
30K
40K
60K
55K
35K
58K
0
100K
200K
300K
400K
Requests per second
These graphs stolen from SmallNest's go framework benchmark.
Search This Blog
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

Μετάφραση Από: Κινέζικα Σε: Αγγλικά

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.



Go News
@golang_news

Ακολουθήστε

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

Προβολή μετάφρασης

RETWEETS 2 ΑΡΕΣΕΙ ΣΕ 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

Features

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

can declare as many as you need*

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

Versioning

Current: **v3.0.0-pre.release**

Read more about Semantic Versioning 2.0.0

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

Install

Compatible with go1.6+

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

this will update the dependencies also.

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

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

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

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

Hi

```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")
    //err := iris.ListenWithErr(":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**

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

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

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

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

Rich Hi with **Django-syntax, flosch/pongo2**

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

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

func main() {
    iris.Config.Render.Template.Engine = iris.PongoEngine
    iris.Get("/hi", hi)
    iris.Listen(":8080")
}

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

- More about configuration here
- More about render and template engines here

TLS

```
// ListenWithErr starts the standalone http server
// which listens to the addr parameter which as the form of
// host:port
//
// It returns an error you are responsible how to handle this
// if you need a func to panic on error use the Listen
// ex: log.Fatal(iris.ListenWithErr(":8080"))
ListenWithErr(addr string) 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 ListenWithErr
// ex: iris.Listen(":8080")
Listen(addr string)

// ListenTLSWithErr 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 returns an error you are responsible how to handle this
// if you need a func to panic on error use the ListenTLS
// ex: log.Fatal(iris.ListenTLSWithErr(":8080","yourfile.cert","
yourfile.key"))
ListenTLSWithErr(addr string, certFile string, keyFile string) e
rror

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

// ListenUNIXWithErr starts the process of listening to the new
requests using a 'socket file', this works only on unix
// returns an error if something bad happens when trying to list
en to
ListenUNIXWithErr(addr string, mode os.FileMode) error

// ListenUNIX starts the process of listening to the new request
s using a 'socket file', this works only on unix
// panics on error
ListenUNIX(addr string, mode os.FileMode

// NoListen is useful only when you want to test Iris, it doesn'
t starts the server but it configures and returns it
NoListen() *Server
```

```
iris.Listen(":8080")
log.Fatal(iris.ListenWithErr(":8080"))

iris.ListenTLS(":8080", "myCERTfile.cert", "myKEYfile.key")
log.Fatal(iris.ListenTLSWithErr(":8080", "myCERTfile.cert", "myK
EYfile.key"))
```

Handlers

Handlers should implement the Handler interface:

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

Using Handlers

```
type myHandlerGet struct {
}

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

//and so on


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

Using HandlerFuncs

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

```
type HandlerFunc func(*Context)

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

HandlerFuncs shoud have this function signature:

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


iris.HandleFunc("GET","/letsgetit",handlerFunc)
//OR
iris.Get("/get", handlerFunc)
iris.Post("/post", handlerFunc)
iris.Put("/put", handlerFunc)
iris.Delete("/delete", handlerFunc)
```

Using Handler API

HandlerAPI is any custom struct which has an `*iris.Context` field.

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

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

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

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

Normally, with HandlerFuncs you should do something like this:

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

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

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

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

But with API you can do this instead:

```
package main

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

```
)

type UserAPI struct {
    *iris.Context
}

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

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

}

// PUT /users
func (u UserAPI) Put() {
    name := u.FormValue("name")
    // myDb.InsertUser(...)
    println(string(name))
    println("Put from /users")
}

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

// DELETE /users/:param1
func (u UserAPI) DeleteBy(id string) {
    // myDb.DeleteUser(id)
```

```
    println("Delete from /" + id)
}

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

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

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

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

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

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

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

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

Using native http.Handler

> Not recommended. Note that using native http handler you cannot access url params.

```
type nativehandler struct {}

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

}

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

Using native http.Handler via iris.ToHandlerFunc()

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

Middleware

Quick view

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

// Then declare which middleware to use (custom or not)
iris.Use(myMiddleware)
iris.UseFunc(myFunc)

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

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

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

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

Handler middleware example:

```
type myMiddleware struct {}

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

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

}

iris.Use(&myMiddleware{})

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

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

HandlerFunc middleware example:

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

iris.UseFunc(myMiddleware)
```

HandlerFunc middleware for a specific route:

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

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

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

> Note that middleware must come before route declaration.

Make use of the middleware, view practical examples here

```go
package main

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

type Page struct {
    Title string
}

iris.Config.Render.Template.Directory = "./yourpath/templates"

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

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

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

API

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

```
package main

import "github.com/kataras/iris"

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

    iris.Listen(":8080")
}

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

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

Declaration

You have wondered this:

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

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

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

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

// 1.
func firstWay() {

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

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

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

```
type (
    // Iris configs for the station
    // All fields can be changed before server's listen except t
he DisablePathCorrection field
```

```
    //
    // MaxRequestBodySize is the only options that can be change
d after server listen -
    // using Config.MaxRequestBodySize = ...
    // Render's rest config can be changed after declaration but
 before server's listen -
    // using Config.Render.Rest...
    // Render's Template config can be changed after declaration
 but before server's listen -
    // using Config.Render.Template...
    // Sessions config can be changed after declaration but befo
re server's listen -
    // using Config.Sessions...
    // and so on...
    Iris struct {

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

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

        // ctx.Param("project") returns the raw named parameter:
 Project%2FDelta
        // which you can escape it manually with net/url:
```

```
        // projectName, _ := url.QueryUnescape(c.Param("project"
).
        // Look here: https://github.com/kataras/iris/issues/135
 for more
        //
        // Default is false
        DisablePathEscape bool

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

        // MaxRequestBodySize Maximum request body size.
        //
        // The server rejects requests with bodies exceeding thi
s limit.
        //
        // By default request body size is -1, unlimited.
        MaxRequestBodySize int64

        // ProfilePath a the route path, set it to enable http p
prof tool
        // Default is empty, if you set it to a $path, these rou
tes will handled:
        // $path/cmdline
        // $path/profile
        // $path/symbol
        // $path/goroutine
        // $path/heap
        // $path/threadcreate
        // $path/pprof/block
        // for example if '/debug/pprof'
        // http://yourdomain:PORT/debug/pprof/
        // http://yourdomain:PORT/debug/pprof/cmdline
        // http://yourdomain:PORT/debug/pprof/profile
        // http://yourdomain:PORT/debug/pprof/symbol
        // http://yourdomain:PORT/debug/pprof/goroutine
        // http://yourdomain:PORT/debug/pprof/heap
        // http://yourdomain:PORT/debug/pprof/threadcreate
```

```
        // http://yourdomain:PORT/debug/pprof/pprof/block
        // it can be a subdomain also, for example, if 'debug.'
        // http://debug.yourdomain:PORT/
        // http://debug.yourdomain:PORT/cmdline
        // http://debug.yourdomain:PORT/profile
        // http://debug.yourdomain:PORT/symbol
        // http://debug.yourdomain:PORT/goroutine
        // http://debug.yourdomain:PORT/heap
        // http://debug.yourdomain:PORT/threadcreate
        // http://debug.yourdomain:PORT/pprof/block
        ProfilePath string

        // Logger the configuration for the logger
        // Iris logs ONLY SEMANTIC errors and the banner if enab
led
        Logger Logger

        // Sessions contains the configs for sessions
        Sessions Sessions

        // Render contains the configs for template and rest con
figuration
        Render Render

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

```go
// 3.
package main

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

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

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

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

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

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

- More about configuration here

Configuration

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

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

Why?

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

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

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

- **easy to remember**: `iris` type takes `config.Iris` , sessions takes `config.Sessions` , `iris.Config.Render` is of type `config.Render` , `iris.Config.Render.Template` is the type `config.Template` , `Logger` takes `config.Logger` and so on...
- **easy to search & find out what features exists and what you can change**: just navigate to the config folder and open the type you want to learn about, for example `/websocket` package's configuration is in `/config/websocket.go`
- **All structs that receive configurations are already set to sane defaults** , so for casual use you can ignore them, but if you ever need to check them, you can find their default configs by this pattern: for example `config.Template` has `config.DefaultTemplate()` , `config.Rest` has `config.DefaultRest()` , `config.Typescript()` has `config.DefaultTypescript()` . Note however that `config.Iris` uses `config.Default()` . Even the plugins have their own default configs, to make it easier for you.

- Enables you to do this **without setting up a config yourself**: `iris.Config.Render.Template.Engine = config.PongoEngine` or `iris.Config.Render.Template.Pongo.Extensions = []string{".xhtml", ".html"}`.
- **(Advanced usage) merge configs**:

```
//...
import "github.com/kataras/iris/config"
//...
templateFromRoutine1 := config.DefaultTemplate()
//....
templateFromOthers := config.Template{ Directory: "views"}

templateConfig := templateFromRoutine1.MergeSingle(templateFromO
thers)

iris.Config.Render.Template = templateConfig
```

All Configs

Party

Let's party with Iris web framework!

```
package main

import "github.com/kataras/iris"

func main() {
    admin := iris.Party("/admin")
    {
        // add a silly middleware
        admin.UseFunc(func(c *iris.Context) {
            //your authentication logic here...
            println("from ", c.PathString())
            authorized := true
            if authorized {
                c.Next()
            } else {
                c.Text(401, c.PathString()+" is not authorized f
or you")
            }

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

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

```
admin/beta/hey") })

    //for subdomains goto: ../subdomains_1/main.go

    iris.Listen(":8080")

}
```

Subdomains

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

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

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

Static

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

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

package main

import "github.com/kataras/iris"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Named Parameters

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

No limit on how long a path can be.

Usage:

```
package main

import (
    "strconv"

    "github.com/kataras/iris"
)

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

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

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

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

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

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

Match anything

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

Static files

Serve a static directory

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

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

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

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

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

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

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

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

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

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

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

Manual static file serving

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

Serve static individual file

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

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

```
package main

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

func main() {

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

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

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

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

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

The previous example is almost identical with

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

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

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

Favicon

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

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

import "github.com/kataras/iris"

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

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

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

Practical example here

Send files

Send a file, force-download to the client

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

```go
package main

import "github.com/kataras/iris"

func main() {

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

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

Send e-mails

This is a package.

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

Configuration

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

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

Example

File: `./main.go`

```
package main
```

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

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

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

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

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

    // standalone

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

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

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

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

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

    // send a body by template
    iris.Get("/send/template", func(ctx *iris.Context) {
        content := iris.TemplateString("body.html", iris.Map{
            "Message": " his is the rich message body sent by a
template!!",
            "Footer":  "The footer of this e-mail!",
        })

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

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

File: `./templates/body.html`

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

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

Render

Click to the headers to open the related doc.

REST

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

Templates

Iris gives you the freedom to render templates through 6 different template engines.

REST

Provides functionality for easily rendering JSON, XML, Text, binary data and Markdown.

config.Rest

```
    // Appends the given character set to the Content-Type heade
r. Default is "UTF-8".
    Charset string
    // Gzip enable it if you want to render with gzip compressio
n. Default is false
    Gzip bool
    // Outputs human readable JSON.
    IndentJSON bool
    // Outputs human readable XML. Default is false.
    IndentXML bool
    // Prefixes the JSON output with the given bytes. Default is
 false.
    PrefixJSON []byte
    // Prefixes the XML output with the given bytes.
    PrefixXML []byte
    // Unescape HTML characters "&<>" to their original values.
Default is false.
    UnEscapeHTML bool
    // Streams JSON responses instead of marshalling prior to se
nding. Default is false.
    StreamingJSON bool
    // Disables automatic rendering of http.StatusInternalServer
Error
    // when an error occurs. Default is false.
    DisableHTTPErrorRendering bool
    // MarkdownSanitize sanitizes the markdown. Default is false.

    MarkdownSanitize bool
```

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

//1.
iris.Config.Render.Rest.IndentJSON = true
iris.Config.Render.Rest...
//2.
restConfig:= config.Rest{
    Charset:                    "UTF-8",
    IndentJSON:                 false,
    IndentXML:                  false,
    PrefixJSON:                 []byte(""),
    PrefixXML:                  []byte(""),
    UnEscapeHTML:               false,
    StreamingJSON:              false,
    DisableHTTPErrorRendering: false,
    MarkdownSanitize: false,
}

iris.Config.Render.Rest = restConfig
```

Usage

The rendering functions simply wraps Go's existing functionality for marshaling and rendering data.

- JSON: Uses the encoding/json package to marshal data into a JSON-encoded response.
- XML: Uses the encoding/xml package to marshal data into an XML-encoded response.
- Binary data: Passes the incoming data straight through to the `iris.Context.Response`.
- Text: Passes the incoming string straight through to the `iris.Context.Response`.

```
package main

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

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

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

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

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

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

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

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

```
            ctx.Markdown(iris.StatusOK, "# Hello Dynamic Markdown
Iris")
        })


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

Templates

Iris gives you the freedom to render templates through **html\/template**, Django-syntax package **Pongo2**, Raw **Markdown**, **Amber**, **Jade** or **Handlebars** via **iris.Config().Render.Template.Engine = iris.___Engine**.

- `iris.HTMLEngine` is the html\/template
- `iris.PongoEngine` is the flosch\/pongo2
- `iris.AmberEngine` is the eknkc\/amber
- `iris.JadeEngine` is the Joker\/jade
- `iris.Handlebars` is the aymerick\/raymond
- `iris.MarkdownEngine`

```
// RenderWithStatus builds up the response from the specified te
mplate and bindings.
RenderWithStatus(status int, name string, binding interface{}, l
ayout ...string) error

// Render same as .RenderWithStatus but with status to iris.Stat
usOK (200)
Render(name string, binding interface{}, layout ...string) error


// TemplateString same as Render but instead of client render, r
eturns the result
TemplateString(name string, binding interface{}, layout ...string
) (string)

// Render same as .Render but
// returns 500 internal server error and logs the error if parse
 failed
MustRender(name string, binding interface{}, layout ...string)
```

A snippet:

```
iris.Get("/default_standar", func(ctx *iris.Context){
  ctx.Render("index.html", nil) // this will render the file ./t
emplates/index.html
})
```

Let's read and learn how to set the configuration now.

```
import (
    "github.com/kataras/iris/config"
    //...
)
// These are the defaults
templateConfig := config.Template {
        Engine:        DefaultEngine, //or HTMLTemplate
        Gzip:          false,
        IsDevelopment: false,
        Directory:     "templates",
        Extensions:    []string{".html"},
        ContentType:   "text/html",
        Charset:       "UTF-8",
        Layout:        "", // currently this is the only config
which not working for pongo2 yet but I will find a way
        HTMLTemplate:  HTMLTemplate{Left: "{{", Right: "}}", Fun
cs: make(map[string]interface{}, 0), LayoutFuncs: make(map[string
]interface{}, 0)},
        Jade:          Jade{Left: "{{", Right: "}}", Funcs: make(
map[string]interface{}, 0), LayoutFuncs: make(map[string]interfa
ce{}, 0)},
        Pongo:         Pongo{Filters: make(map[string]pongo2.Fil
terFunction, 0), Globals: make(map[string]interface{}, 0)},
        Markdown:      Markdown{Sanitize: false},
        Amber:         Amber{Funcs: template.FuncMap{}},
        Handlebars:    Handlebars{Helpers: make(map[string]inter
face{}, 0)},
}

// Set
```

```
// 1. Directly via complete custom configuration field
iris.Config.Render.Template = templateConfig

// 2. Fast way - Pongo snippet
iris.Config.Render.Template.Engine = iris.PongoEngine
iris.Config.Render.Template.Directory = "mytemplates"
iris.Config.Render.Template.Pongo.Filters = ...

// 3. Fast way - HTMLTemplate snippet
iris.Config.Render.Template.Engine = iris.HTMLTemplate // or iris.DefaultEngine
iris.Config.Render.Template.Layout = "layout/layout.html" // = ./templates/layout/layout.html
//...

// 4.
theDefaults := config.DefaultTemplate()
theDefaults.Extensions = []string{".myExtension"}
//...
```

Examples

`HTMLTemplate`

```go
// main.go

package main

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

type mypage struct {
  Message string
}

func main() {
    iris.Config.Render.Template.Layout = "layouts/layout.html" /
/ default ""
    iris.Get("/", func(ctx *iris.Context) {
         ctx.MustRender("page1.html", mypage{"Message from page1
!"})
    })

    println("Server is running at: 8080")
    iris.Listen(":8080")
}
```

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

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

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

```
<!-- templates/page1.html -->

<div style="background-color:black;color:blue">

<h1> The message: {{.Message}} </h1>

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

</div>
```

```
<!-- templates/partials/page1_partial1.html -->

<div style="background-color:white;color:red"> <h1> Page 1's Par
tial 1 </h1> </div>
```

Run main.go open browser and navigate to the localhost:8080 -> view page source, this is the **output**:

```
<!-- OUTPUT -->
<html>
  <head>
    <title>My Layout</title>
  </head>
  <body>

    <div style="background-color:black;color:blue">

    <h1> The message: Message from page1! </h1>

    <div style="background-color:white;color:red">
        <h1> Page 1's Partial 1 </h1> </div>
    </div>

  </body>
</html>
```

Pongo

```go
// main.go
package main

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

func main() {

    iris.Config.Render.Template.Engine = config.PongoEngine // o
r iris.PongoEngine without need to import the config

    iris.Get("/", func(ctx *iris.Context) {
        ctx.MustRender("index.html", map[string]interface{}{"use
rname": "iris", "is_admin": true})
    })

    println("Server is running at :8080")
    iris.Listen(":8080")
}
```

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

<html>
<head><title>Hello Pongo2 from Iris</title></head>
<body>
     {% if is_admin %}<p>{{username}} is an admin!</p>{% endif %
}
</body>
</html>
```

Run main.go open browser and navigate to the localhost:8080 -> view page source, this is the **output**:

```
<!-- OUTPUT -->
<html>
<head><title>Hello Pongo2 from Iris</title></head>
<body>
     <p>iris is an admin!</p>
</body>
</html>
```

Markdown

```
// main.go
package main

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

func main() {
    // Markdown engine doesn't supports Layout and context bindi
ng
    iris.Config.Render.Template.Engine = config.MarkdownEngine
    iris.Config.Render.Template.Extensions = []string{".md"}

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

        err := ctx.Render("index.md", nil) // doesnt' supports a
ny context binding, just pure markdown
        if err != nil {
            panic(err)
        }
    })

    println("Server is running at :8080")
    iris.Listen(":8080")
}
```

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

```
## 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
.
```

Run main.go open browser and navigate to the localhost:8080 -> view page source, this is the **output**:

```
<!-- OUTPUT -->

<h2>Hello Markdown from Iris</h2>

<p>This is an example of Markdown with Iris</p>

<h2>Features</h2>

<p>All features of Sundown are supported, including:
* <strong>Compatibility</strong>. The Markdown v1.0.3 test suite
 passes with
the <code>--tidy</code> option. Without <code>--tidy</code>, the
 differences are
mostly in whitespace and entity escaping, where blackfriday is
more consistent and cleaner.
* <strong>Common extensions</strong>, including table support, f
enced code
blocks, autolinks, strikethroughs, non-strict emphasis, etc.
* <strong>Safety</strong>. 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: &ldquo;safety&rdquo; in this context means <em>runtime saf
ety only</em>. In order to
protect yourself against JavaScript injection in untrusted conte
nt, see
<a href="https://github.com/russross/blackfriday#sanitize-untrus
ted-content">this example</a>.
* <strong>Fast processing</strong>. It is fast enough to render
on-demand in
most web applications without having to cache the output.
* <strong>Thread safety</strong>. You can run multiple parsers i
n different
goroutines without ill effect. There is no dependence on global
shared state.
* <strong>Minimal dependencies</strong>. Blackfriday only depend
s 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.
* <strong>Standards compliant</strong>. Output successfully vali
dates using the
W3C validation tool for HTML 4.01 and XHTML 1.0 Transitional.</p>
```

Amber

```go
// main.go
package main

import "github.com/kataras/iris"

func main() {

    iris.Config.Render.Template.Engine = iris.AmberEngine
    iris.Config.Render.Template.Extensions = []string{".amber"}
    // this is optionally, you can just leave it to default whic
h is .html

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Render("basic.amber", map[string]string{"Name": "iri
s"})

    })

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

```
<!-- 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
```

Run main.go open browser and navigate to the localhost:8080 -> view page source, this is the **output**:

```
<!-- OUTPUT -->
<!DOCTYPE html>
<html>
    <head>
        <title>Hello Amber from Iris</title>
        <meta name="description" value="This is a sample" />
        <script type="text/javascript">
            var hw = "Hello iris!"
            alert(hw)
        </script>
        <style type="text/css">
            body {
                background: maroon;
                color: white
            }
        </style>
    </head>
    <body>
        <header id="mainHeader">
            <ul>
                <li class="active">
                    <a href="/" title="Main Page">Main Page</a>
                </li>
            </ul>
            <h1>Hi iris</h1>
        </header>
        <footer>
            Hey
            <br />
            There
        </footer>
    </body>
</html>
```

Jade

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

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

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

type Job struct {
    Employer string
    Role     string
}

func main() {
    iris.Config.Render.Template.Extensions = []string{".jade"}
    // this is optionally, you can keep .html extension
    iris.Config.Render.Template.Engine = iris.JadeEngine

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

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

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

        ctx.MustRender("page.jade", person)
    })

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

```
}
```

```
<!-- templates/page.jade -->

doctype html
html(lang=en)
    head
        meta(charset=utf-8)
        title Title
    body
        p ads
        ul
            li The name is {{.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}}
```

Run main.go open browser and navigate to the localhost:8080 -> view page source, this is the **output**:

```
<!-- OUTPUT -->

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Title</title>
    </head>
    <body>
        <p>ads</p>
        <ul>
            <li>The name is name1.</li>
            <li>The age is 50.</li>
        </ul>

            <div>An email is email1@something.gr</div>

            <div>An email is email2.anything@gmail.com</div>



                <div>
                 An employer is Super Employer
                 and the role is Team leader
                </div>

                <div>
                 An employer is Fast Employer
                 and the role is Project managment
                </div>


    </body>
</html>
```

Handlebars

For a more complete example with party, no layout, different layouts and partials go here.

```
// main.go
//Package main a basic and simple example on how to use handleba
rs with Iris

package main

import (
    "github.com/aymerick/raymond"

    "github.com/kataras/iris"
)

func main() {
    // set the template engine
    iris.Config.Render.Template.Engine = iris.HandlebarsEngine

    // optionaly set handlebars helpers by importing "github.com
/aymerick/raymond" when you need to return and render html
    iris.Config.Render.Template.Handlebars.Helpers["boldme"] = f
unc(input string) raymond.SafeString {
        return raymond.SafeString("<b> " + input + "</b>")
    }

    // NOTE:
    // the Iris' route framework {{url "my-routename" myparams}}
 and {{urlpath "my-routename" myparams}} are working like all ot
her template engines,
    // so  avoid custom url and urlpath helpers.

    iris.Get("/", func(ctx *iris.Context) {
        // optionally, set a context  for the template
        mycontext := iris.Map{"Name": "Iris", "Type": "Web"}

        ctx.Render("home.html", mycontext)
    })
    iris.Listen(":8080")
}

/*
```

```
MORE DOCS CAN BE FOUND HERE: https://github.com/aymerick/raymond
*/
```

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

<html>
  <head>
    <title>My Home page</title>

  </head>
  <body>
    Name: {{boldme Name}} <br/>
    Type: {{boldme Type}}
  </body>
</html>
```

Run main.go open browser and navigate to the localhost:8080 -> view page source, this is the **output**:

```
<!-- OUTPUT -->

<html>
  <head>
    <title>My Home page</title>

  </head>
  <body>
    Name: <b> Iris</b> <br/>
    Type: <b> Web</b>
  </body>
</html>
```

Gzip

Gzip compression is easy.

For **auto-gzip** to all rest and template responses, look the Gzip option at the iris.Config().Render.Rest.Gzip and iris.Config().Render.Template.Gzip here

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

WriteGzip(w *bufio.Writer) error


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

How to use

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

Other

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

Streaming

Do progressive rendering via multiple flushes, streaming.

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

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

Usage example

```
package main

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

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

    iris.Listen(":8080")
}

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

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

To achieve the oposite make use of the `StreamReader`

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

Cookies

Cookie management, even your little brother can do this!

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

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

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

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

How to use

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

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

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

Flash messages

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

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

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

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

Example

```
package main

import (
```

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

func main() {

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

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

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

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

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

    })

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

Body binder

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

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

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

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

How to use

JSON

```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

Custom Marshaling

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

Example

```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")
}
```

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

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

Example

In form html

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

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

Backend

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

```
package main

type InterfaceStruct struct {
    ID   int
    Name string
}

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

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

  Interface interface{}
}

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

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

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

Custom HTTP Errors

You can define your own handlers when http error occurs.

```
package main

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

func main() {

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

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

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

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

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

}
```

Context

```
type (
    // IContext the interface for the Context
    IContext interface {
        IContextRenderer
        IContextStorage
        IContextBinder
        IContextRequest
        IContextResponse
        SendMail(string, string, ...string) error
        Log(string, ...interface{})
        Reset(*fasthttp.RequestCtx)
        GetRequestCtx() *fasthttp.RequestCtx
        Clone() IContext
        Do()
        Next()
        StopExecution()
        IsStopped() bool
        GetHandlerName() string
    }

    // IContextBinder is part of the IContext
    IContextBinder interface {
        ReadJSON(interface{}) error
        ReadXML(interface{}) error
        ReadForm(formObject interface{}) error
    }

    // IContextRenderer is part of the IContext
    IContextRenderer interface {
        Write(string, ...interface{})
        HTML(int, string)
        // Data writes out the raw bytes as binary data.
        Data(status int, v []byte) error
        // RenderWithStatus builds up the response from the spec
ified template and bindings.
```

```
        RenderWithStatus(status int, name string, binding interf
ace{}, layout ...string) error
        // Render same as .RenderWithStatus but with status to i
ris.StatusOK (200)
        Render(name string, binding interface{}, layout ...string
) error
        // MustRender same as .Render but returns 500 internal s
erver http status (error) if rendering fail
        MustRender(name string, binding interface{}, layout ...s
tring)
        // TemplateString accepts a template filename, its conte
xt data and returns the result of the parsed template (string)
        // if any error returns empty string
        TemplateString(name string, binding interface{}, layout
...string) string
        // MarkdownString parses the (dynamic) markdown string a
nd returns the converted html string
        MarkdownString(markdown string) string
        // Markdown parses and renders to the client a particula
r (dynamic) markdown string
        // accepts two parameters
        // first is the http status code
        // second is the markdown string
        Markdown(status int, markdown string)
        // JSON marshals the given interface object and writes t
he JSON response.
        JSON(status int, v interface{}) error
        // JSONP marshals the given interface object and writes
the JSON response.
        JSONP(status int, callback string, v interface{}) error
        // Text writes out a string as plain text.
        Text(status int, v string) error
        // XML marshals the given interface object and writes th
e XML response.
        XML(status int, v interface{}) error

        ExecuteTemplate(*template.Template, interface{}) error
        ServeContent(io.ReadSeeker, string, time.Time, bool) err
or
        ServeFile(string, bool) error
```

```
        SendFile(filename string, destinationName string) error
        Stream(func(*bufio.Writer))
        StreamWriter(cb func(writer *bufio.Writer))
        StreamReader(io.Reader, int)
    }

    // IContextRequest is part of the IContext
    IContextRequest interface {
        Param(string) string
        ParamInt(string) (int, error)
        URLParam(string) string
        URLParamInt(string) (int, error)
        URLParams() map[string]string
        MethodString() string
        HostString() string
        Subdomain() string
        PathString() string
        RequestPath(bool) string
        RequestIP() string
        RemoteAddr() string
        RequestHeader(k string) string
        PostFormValue(string) string
        // PostFormMulti returns a slice of string from post req
uest's data
        PostFormMulti(string) []string
    }

    // IContextResponse is part of the IContext
    IContextResponse interface {
        // SetStatusCode sets the http status code
        SetStatusCode(int)
        // SetContentType sets the "Content-Type" header, receiv
es the value
        SetContentType(string)
        // SetHeader sets the response headers first parameter i
s the key, second is the value
        SetHeader(string, string)
        Redirect(string, ...int)
        RedirectTo(routeName string, args ...interface{})
        // Errors
```

```
        NotFound()
        Panic()
        EmitError(int)
        //
    }

    // IContextStorage is part of the IContext
    IContextStorage interface {
        Get(string) interface{}
        GetString(string) string
        GetInt(string) int
        Set(string, interface{})
        SetCookie(*fasthttp.Cookie)
        SetCookieKV(string, string)
        RemoveCookie(string)
        // Flash messages
        GetFlash(string) string
        GetFlashBytes(string) ([]byte, error)
        SetFlash(string, string)
        SetFlashBytes(string, []byte)
        Session() store.IStore
        SessionDestroy()
    }
)
```

The examples will give you the direction.

Logger

This is a middleware

Logs the incoming requests

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

How to use

```go
package main

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

/*
With configs:

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

iris.Use(errorLogger)
```

```
With default configs:

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

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

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

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

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

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


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

    iris.Listen(":8080")

}
```

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

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

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

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

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

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

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

        // foreground colors single SGR Code

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

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

        // background colors single SGR Code

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

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

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

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

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

HTTP access control

This is a middleware.

Some security work for you between the requests.

Options

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

```
    AllowedHeadersAll bool

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

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

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

Example

```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

```
package main

import (
    "time"

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

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

    authentication := basicauth.New(authConfig)

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

    // to party

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

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

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

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

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

OAuth, OAuth2

This is a plugin.

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

Supported Providers

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

How to use - high level

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

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

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

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

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

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

Example:

```
// main.go
package main

import (
    "sort"
    "strings"
```

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

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

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

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

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

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

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

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

    var keys []string
```

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

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

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

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

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

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

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

View:

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

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

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

How to use - low level

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

Example:

```
package main

import (
    "html/template"
    "os"

    "sort"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    })

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

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

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

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

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

high level and low level, no performance differences

JSON Web Tokens

This is a middleware.

What is it?

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

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

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

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

Example

```
package main

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

func main() {

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

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

    iris.Get("/ping", PingHandler)

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

}

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

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

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

Secure

This is a middleware

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

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

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

Example

```
package main

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

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


        // AllowedHosts is a list of fully qualified domain names

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

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

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

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

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

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


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


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

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

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

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

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

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

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

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

        c.Next()
    })

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

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

Sessions

This is a package

This package is new and unique, if you notice a bug or issue post it here

- Cleans the temp memory when a sessions is iddle, and re-loccate it , fast, to the temp memory when it's necessary. Also most used/regular sessions are going front in the memory's list.
- Supports redisstore and normal memory routing. If redisstore is used but fails to connect then ,automatically, switching to the memory storage.

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

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

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

How to use - easy way

Example **memory**

```
package main

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

```
)

func main() {


    // when import _ "github.com/kataras/iris/sessions/providers
/memory"
    //iris.Config.Sessions.Provider = "memory"
    // The cookie name
    //iris.Config.Sessions.Cookie = "irissessionid"
    // Expires the date which the cookie must expires. Default i
nfinitive/unlimited life (config.CookieExpireNever)
    //iris.Config.Sessions.Expires = time.Time....
    // GcDuration every how much duration(GcDuration) the memory
 should be clear for unused cookies
    //iris.Config.Sessions.GcDuration = time.Duration(2) *time.H
our

    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) {
        name := c.Session().GetString("name")

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

    iris.Get("/delete", func(c *iris.Context) {
        //get the session for this context

        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()
    })

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

Example default **redis**

```
package main

import (
    "github.com/kataras/iris"
     _ "github.com/kataras/iris/sessions/providers/redis"
)

func main() {

    iris.Config.Sessions.Provider = "redis"

    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) {
        name := c.Session().GetString("name")

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

    iris.Get("/delete", func(c *iris.Context) {
        //get the session for this context

        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()
    })

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

Example customized **config.Redis**

```
    // Redis the redis configuration used inside sessions
    Redis struct {
        // Network "tcp"
        Network string
        // Addr "127.0.01:6379"
        Addr string
        // Password string .If no password then no 'AUTH'. Defau
lt ""
        Password string
        // If Database is empty "" then no 'SELECT'. Default ""
        Database string
        // MaxIdle 0 no limit
        MaxIdle int
        // MaxActive 0 no limit
        MaxActive int
        // IdleTimeout  time.Duration(5) * time.Minute
        IdleTimeout time.Duration
        // Prefix "myprefix-for-this-website". Default ""
        Prefix string
        // MaxAgeSeconds how much long the redis should keep
        // the session in seconds. Default 31556926.0 (1 year)
        MaxAgeSeconds int
    }
```

```go
package main

import (
    "github.com/kataras/iris"
    "github.com/kataras/iris/sessions/providers/redis"
)

func init() {
    redis.Config.Addr = "127.0.0.1:2222"
    redis.Config.MaxAgeSeconds = 5000.0
}

func main() {
```

```
    iris.Config.Sessions.Provider = "redis"

    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) {
        name := c.Session().GetString("name")

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

    iris.Get("/delete", func(c *iris.Context) {
        //get the session for this context

        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()
    })

    println("Server is listening at :8080")
    iris.Listen("8080")
}
```

How to use - hard way

```
// New creates & returns a new Manager and start its GC
// accepts 4 parameters
// first is the providerName (string) ["memory","redis"]
// second is the cookieName, the session's name (string) ["myses
sionsecretcookieid"]
// third is the gcDuration (time.Duration)
// when this time passes it removes from
// temporary memory GC the value which hasn't be used for a long
 time(gcDuration)
// this is for the client's/browser's Cookie life time(expires)
also

New(provider string, cName string, gcDuration time.Duration) *se
ssions.Manager
```

Example **memory**

```
package main

import (
    "time"

    "github.com/kataras/iris"
    "github.com/kataras/iris/config"
    "github.com/kataras/iris/sessions"

    _ "github.com/kataras/iris/sessions/providers/memory"
)

var sess *sessions.Manager

func init() {
    sessConfig := config.Sessions{
        Provider:   "memory", // if you set it to ""  means that
```

```
 sessions are disabled.
        Cookie:     "yoursessionCOOKIEID",
        Expires:    config.CookieExpireNever,
        GcDuration: time.Duration(2) * time.Hour,
    }
    sess = sessions.New(sessConfig) // or just sessions.New()
}

func main() {

    iris.Get("/set", func(c *iris.Context) {
        //get the session for this context
        session := sess.Start(c)

        //set session values
        session.Set("name", "kataras")

        //test if setted here
        c.Write("All ok session setted to: %s", session.Get("nam
e"))
    })

    iris.Get("/get", func(c *iris.Context) {
        //get the session for this context
        session := sess.Start(c)

        var name string

        //get the session value
        if v := session.Get("name"); v != nil {
            name = v.(string)
        }
        // OR just name = session.GetString("name")

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

        iris.Get("/delete", func(c *iris.Context) {
        //get the session for this context
        session := sess.Start(c)
```

```
        session.Delete("name")

    })

    iris.Get("/clear", func(c *iris.Context) {
        //get the session for this context
        session := sess.Start(c)
        // removes all entries
        session.Clear()
    })

    iris.Get("/destroy", func(c *iris.Context) {
        //destroy, removes the entire session and cookie
        sess.Destroy(c)
    })

    iris.Listen(":8080")
}

// session.GetAll() returns all values a map[interface{}]interfa
ce{}
// session.VisitAll(func(key interface{}, value interface{}) { /
* loops for each entry */})

}
```

Example **redis** with config.Redis defaults

The default redis client points to 127.0.0.1:6379

```go
package main

import (
    "time"

    "github.com/kataras/iris"
    "github.com/kataras/iris/config"
    "github.com/kataras/iris/sessions"

    _ "github.com/kataras/iris/sessions/providers/redis"
)

var sess *sessions.Manager

func init() {
    sessConfig := config.Sessions{
          Provider:   "redis",
          Cookie:     "yoursessionCOOKIEID",
          Expires:    config.CookieExpireNever,
          GcDuration: time.Duration(2) * time.Hour,
    }

    sess := sessions.New(sessConfig)
}

//... usage: same as memory
```

Example **redis** with custom configuration **config.Redis**

```
    // Redis the redis configuration used inside sessions
    Redis struct {
        // Network "tcp"
        Network string
        // Addr "127.0.01:6379"
        Addr string
        // Password string .If no password then no 'AUTH'. Defau
lt ""
        Password string
        // If Database is empty "" then no 'SELECT'. Default ""
        Database string
        // MaxIdle 0 no limit
        MaxIdle int
        // MaxActive 0 no limit
        MaxActive int
        // IdleTimeout  time.Duration(5) * time.Minute
        IdleTimeout time.Duration
        // Prefix "myprefix-for-this-website". Default ""
        Prefix string
        // MaxAgeSeconds how much long the redis should keep
        // the session in seconds. Default 31556926.0 (1 year)
        MaxAgeSeconds int
    }
```

```go
package main

import (
    "time"

    "github.com/kataras/iris"
    "github.com/kataras/iris/config"
    "github.com/kataras/iris/sessions"

     "github.com/kataras/iris/sessions/providers/redis"
)

var sess *sessions.Manager

func init() {
    // you can config the redis after init also, but before any
client's request
    // but it's always a good idea to do it before sessions.New.
..
    redis.Config.Network = "tcp"
    redis.Config.Addr = "127.0.0.1:6379"
    redis.Config.Prefix = "myprefix-for-this-website"

    sessConfig := config.Sessions{
          Provider:   "redis",
          Cookie:     "yoursessionCOOKIEID",
          Expires:    config.CookieExpireNever,
          GcDuration: time.Duration(2) * time.Hour,
    }

    sess := sessions.New(sessConfig)
}

//...usage: same as memory
```

Security: Prevent session hijacking

> This section is external

cookie only and token

Through this simple example of hijacking a session, you can see that it's very dangerous because it allows attackers to do whatever they want. So how can we prevent session hijacking?

The first step is to only set session ids in cookies, instead of in URL rewrites. Also, Iris has already set the httponly cookie property to true. This restricts client side scripts that want access to the session id. Using these techniques, cookies cannot be accessed by XSS and it won't be as easy as we showed to get a session id from a cookie manager.

The second step is to add a token to every request. Similar to the way we dealt with repeat forms in previous sections, we add a hidden field that contains a token. When a request is sent to the server, we can verify this token to prove that the request is unique.

```
h := md5.New()
salt:="secretkey%^7&8888"
io.WriteString(h,salt+time.Now().String())
token:=fmt.Sprintf("%x",h.Sum(nil))
if r.Form["token"]!=token{
    // ask to log in
}
session.Set("token",token)
```

Session id timeout

Another solution is to add a create time for every session, and to replace expired session ids with new ones. This can prevent session hijacking under certain circumstances.

```
createtime := session.Get("createtime")
if createtime == nil {
    session.Set("createtime", time.Now().Unix())
} else if (createtime.(int64) + 60) < (time.Now().Unix()) {
    sess.Destroy(c)
    session = sess.Start(c)
}
```

We set a value to save the create time and check if it's expired (I set 60 seconds here). This step can often thwart session hijacking attempts.

Combine the two solutions above and you will be able to prevent most session hijacking attempts from succeeding. On the one hand, session ids that are frequently reset will result in an attacker always getting expired and useless session ids; on the other hand, by already setted the httponly property on cookies and ensuring that session ids can only be passed via cookies, all URL based attacks are mitigated.

Websockets

This is a package

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

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

Read more about Websockets

Configuration

```
type Websocket struct {
    // WriteTimeout time allowed to write a message to the conne
ction.
    // Default value is 10 * 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 int
    // Endpoint is the path which the websocket server will list
en for clients/connections
    // Default value is empty string, if you don't set it the We
bsocket server is disabled.
    Endpoint string
    // Headers  the response headers before upgrader
    // Default is empty
    Headers map[string]string
}
```

```
iris.Config().Websocket
```

Outline

websocket.Server / iris.Websocket

```
OnConnection(func(c websocket.Connection){})
```

websocket.Connection

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

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

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

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

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

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

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

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


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

How to use

Server-side

```
package main

import (
    "fmt"

    "github.com/kataras/iris"
    "github.com/kataras/iris/websocket"
)

type clientPage struct {
    Title string
    Host  string
}

func main() {

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

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

    // the path which the websocket client should listen/registe
d to ->
    iris.Config.Websocket.Endpoint = "/my_endpoint"

    ws := iris.Websocket // get the websocket server

    var myChatRoom = "room1"
    ws.OnConnection(func(c websocket.Connection) {

        c.Join(myChatRoom)

        c.On("chat", func(message string) {
```

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

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

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

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

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

Client-side

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

$(function () {

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


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

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

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

    $("#sendBtn").click(function () {
        //ws.EmitMessage(messageTxt.val());
        ws.Emit("chat", messageTxt.val().toString());
        messageTxt.val("");
    })

})


function appendMessage(messageDiv) {
    var theDiv = messages[0]
    var doScroll = theDiv.scrollTop == theDiv.scrollHeight - the
Div.clientHeight;
    messageDiv.appendTo(messages)
    if (doScroll) {
        theDiv.scrollTop = theDiv.scrollHeight - theDiv.clientHe
ight;
    }
}
```

```
<html>

<head>
    <title>My iris-ws</title>
</head>

<body>
    <div id="messages" style="border-width:1px;border-style:soli
d;height:400px;width:375px;">

    </div>
    <input type="text" id="messageTxt" />
    <button type="button" id="sendBtn">Send</button>
    <script type="text/javascript">
        var HOST = {{.Host}}
    </script>
    <script src="js/vendor/jquery-2.2.3.min.js" type="text/javas
cript"></script>
    <!-- /iris-ws.js is served automatically by the server -->
    <script src="/iris-ws.js" type="text/javascript"></script>
    <!-- -->
    <script src="js/chat.js" type="text/javascript"></script>
</body>

</html>
```

View a working example by navigating here and if you need more than one websocket server click here

Graceful

This is a package.

Enables graceful shutdown.

```
package main

import (
    "time"
    "github.com/kataras/iris"
    "github.com/iris-contrib/graceful"
)

func main() {
    api := iris.New()
    api.Get("/", func(c *iris.Context) {
        c.Write("Welcome to the home page!")
    })

    graceful.Run(":3001", time.Duration(10)*time.Second, api)
}
```

Recovery

This is a middleware.

Safety recover the server from panic.

```
recovery.New(...io.Writer)
```

```go
package main

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

func main() {

    iris.Use(recovery.New(os.Stderr)) // optional

    iris.Get("/", func(ctx *iris.Context) {
        ctx.Write("Hi, let's panic")
        panic("Something bad!")
    })

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

Plugins

Plugins are modules that you can build to inject the Iris' flow. Think it like a middleware for the Iris framework itself, not the requests. Middleware starts it's actions after the server listen and executes itself on every request, Plugin on the other hand starts working when you registered it, it has to do with framework's code, it has access to the *iris.Framework, so it can register routes, start a second server, read the iris' configs or edit them and all things you can do with Iris. Look how it's interface looks:

```
type (
    // Plugin just an empty base for plugins
    // A Plugin can be added with: .Add(PreListenFunc(func(*Fram
ework))) and so on... or
    // .Add(myPlugin{},myPlugin2{}) which myPlugin is  a struct
with any of the methods below or
    //.PostListen(func(*Framework)) and so on...
    Plugin interface {
    }

    // pluginGetName implements the GetName() string method
    pluginGetName interface {
        // GetName has to returns the name of the plugin, a name
 is unique
        // name has to be not dependent from other methods of th
e plugin,
        // because it is being called even before the Activate
        GetName() string
    }

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

```
    // pluginActivate implements the Activate(pluginContainer) e
rror method
    pluginActivate interface {
        // Activate called BEFORE the plugin being added to the
plugins list,
        // if Activate returns none nil error then the plugin is
 not being added to the list
        // it is being called only one time
        //
        // PluginContainer parameter used to add other plugins i
f that's necessary by the plugin
        Activate(PluginContainer) error
    }
    // pluginPreListen implements the PreListen(*Framework) meth
od
    pluginPreListen interface {
        // PreListen it's being called only one time, BEFORE the
 Server is started (if .Listen called)
        // is used to do work at the time all other things are r
eady to go
        //  parameter is the station
        PreListen(*Framework)
    }
    // PreListenFunc implements the simple function listener for
 the PreListen(*Framework)
    PreListenFunc func(*Framework)
    // pluginPostListen implements the PostListen(*Framework) me
thod
    pluginPostListen interface {
        // PostListen it's being called only one time, AFTER the
 Server is started (if .Listen called)
        // parameter is the station
        PostListen(*Framework)
    }
    // PostListenFunc implements the simple function listener fo
r the PostListen(*Framework)
    PostListenFunc func(*Framework)
    // pluginPreClose implements the PreClose(*Framework) method
    pluginPreClose interface {
```

```
        // PreClose it's being called only one time, BEFORE the
Iris .Close method
        // any plugin cleanup/clear memory happens here
        //
        // The plugin is deactivated after this state
        PreClose(*Framework)
    }
    // PreCloseFunc implements the simple function listener for
the PreClose(*Framework)
    PreCloseFunc func(*Framework)

    // pluginPreDownload It's for the future, not being used, I
need to create
    // and return an ActivatedPlugin type which will have it's m
ethods, and pass it on .Activate
    // but now we return the whole pluginContainer, which I can'
t determinate which plugin tries to
    // download something, so we will leave it here for the futu
re.
    pluginPreDownload interface {
        // PreDownload it's being called every time a plugin tri
es to download something
        //
        // first parameter is the plugin
        // second parameter is the download url
        // must return a boolean, if false then the plugin is no
t permmited to download this file
        PreDownload(plugin Plugin, downloadURL string) // bool
    }

    // PreDownloadFunc implements the simple function listener f
or the PreDownload(plugin,string)
    PreDownloadFunc func(Plugin, string)

)
```

```
package main

import (
```

```
    "fmt"

    "github.com/kataras/iris"
)

func main() {
    // first way:
    // simple way for simple things
    // PreListen before a station is listening ( iris.Listen/TLS
...)
    iris.Plugins.PreListen(func(s *iris.Framework) {
        for _, route := range s.Lookups() {
            fmt.Printf("Func: Route Method: %s | Subdomain %s |
Path: %s is going to be registed with %d handler(s). \n", route.
Method(), route.Subdomain(), route.Path(), len(route.Middleware(
)))
        }
    })

    // second way:
    // structured way for more things
    plugin := myPlugin{}
    iris.Plugins.Add(plugin)

    iris.Get("/first_route", aHandler)

    iris.Post("/second_route", aHandler)

    iris.Put("/third_route", aHandler)

    iris.Get("/fourth_route", aHandler)

    iris.Listen(":8080")
}

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

type myPlugin struct{}
```

```
// PostListen after a station is listening ( iris.Listen/TLS...)
func (pl myPlugin) PostListen(s *iris.Framework) {
    fmt.Printf("myPlugin: server is listening on host: %s", s.HT
TPServer.Host())
}

//list:
/*
    Activate(iris.PluginContainer)
    GetName() string
    GetDescription() string
    PreListen(*iris.Framework)
    PostListen(*iris.Framework)
    PreClose(*iris.Framework)
    PreDownload(thePlugin iris.Plugin, downloadUrl string)
    // for custom events
    On(string,...func())
    Call(string)
*/
```

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

Take a look at the real plugins, easy to make your own.

Internationalization and Localization

This is a middleware

Tutorial

Create folder named 'locales'

```
///Files:

./locales/locale_en-US.ini
./locales/locale_el-US.ini
```

Contents on locale_en-US:

```
hi = hello, %s
```

Contents on locale_el-GR:

```
hi = Γειά, %s
```

```
package main

import (
    "fmt"
    "github.com/kataras/iris"
    "github.com/iris-contrib/middleware/i18n"
)

func main() {

    iris.Use(i18n.New(i18n.Config{Default: "en-US",
        Languages: map[string]string{
            "en-US": "./locales/locale_en-US.ini",
            "el-GR": "./locales/locale_el-GR.ini",
            "zh-CN": "./locales/locale_zh-CN.ini"}}))

    iris.Get("/", func(ctx *iris.Context) {
        hi := ctx.GetFmt("translate")("hi", "maki") // hi is
 the key, 'maki' is the %s, the second parameter is optional
        language := ctx.Get("language") // language is the l
anguage key, example 'en-US'

        ctx.Write("From the language %s translated output: %
s", language, hi)
    })

    iris.Listen(":8080")

}
```

Typescript

This is a plugin.

This is an Iris and typescript bridge plugin.

What?

1. Search for typescript files (.ts)
2. Search for typescript projects (.tsconfig)
3. If 1 || 2 continue else stop
4. Check if typescript is installed, if not then auto-install it (always inside npm global modules, -g)
5. If typescript project then build the project using tsc -p $dir
6. If typescript files and no project then build each typescript using tsc $filename
7. Watch typescript files if any changes happens, then re-build (5|6)

> Note: Ignore all typescript files & projects whose path has '/node_modules/'

Options

- **Bin**: string, the typescript installation path/bin/tsc or tsc.cmd, if empty then it will search to the global npm modules
- **Dir**: string, Dir set the root, where to search for typescript files/project. Default "./"
- **Ignore**: string, comma separated ignore typescript files/project from these directories. Default "" (node_modules are always ignored)
- **Tsconfig**: config.Tsconfig{}, here you can set all compilerOptions if no tsconfig.json exists inside the 'Dir'
- **Editor**: config.Typescript { Editor: config.Editor{}, if setted then alm-tools browser-based typescript IDE will be available. Defailt is nil

> All these are optional

How to use

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/plugin/typescript"
)

func main(){
    ts := typescript.Config {
        Dir: "./scripts/src",
        Tsconfig: typescript.Tsconfig{Module: "commonjs", Target
: "es5"},
    }
    // or typescript.DefaultConfig()

    iris.Plugins.Add(typescript.New(ts)) //or with the default o
ptions just: typescript.New()

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

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

Enable web browser editor

```
ts := typescript.Typescript {
    //...
    Editor: typescript.Editor{Username:"admin", Password: "admin
!123"}
    //...
}
```

Editor

This is a plugin.

Editor Plugin is just a bridge between Iris and alm-tools.

alm-tools is a typescript online IDE/Editor, made by @basarat one of the top contributors of the Typescript.

Iris gives you the opportunity to edit your client-side using the alm-tools editor, via the editor plugin.

This plugin starts it's own server, if Iris server is using TLS then the editor will use the same key and cert.

How to use

```
package main

import (
    "github.com/kataras/iris"
    "github.com/iris-contrib/plugin/editor"
)

func main(){
    e := editor.New()
   // editor.Config{ Username: "admin", Password: "admin!123", P
ort: 4444, WorkingDir: "/public/scripts"}

    iris.Plugins.Add(e)

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

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

Note for username, password: The Authorization specifies the authentication mechanism (in this case Basic) followed by the username and password. Although, the string aHR0cHdhdGNoOmY= may look encrypted it is simply a base64 encoded version of username:password. Would be readily available to anyone who could intercept the HTTP request. Read more here.

> The editor can't work if the directory doesn't contains a tsconfig.json.
>
> If you are using the typescript plugin you don't have to call the .Dir(...)

Control panel

This is a plugin which is working but not finished yet.

Which gives access to your iris server's information via a web interface.

> You need internet connection the first time you will run this plugin, because the assets don't exists to this repository but here. The plugin will install these for you at the first run.

How to use

```
iriscontrol.New(port int, authenticatedUsers map[string]string)
iris.IPlugin
```

Example

```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")
}
```