

# Iris web framework



The complete guide

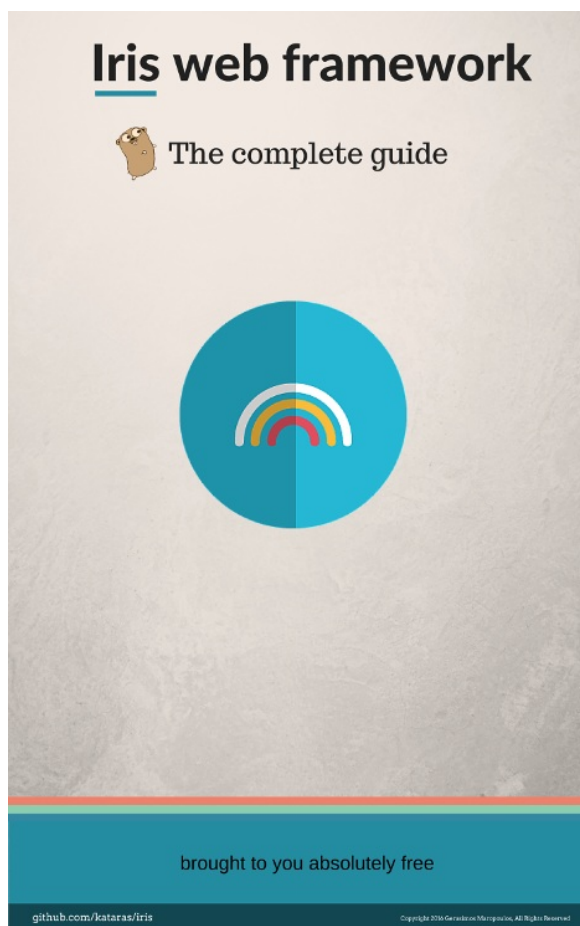


brought to you absolutely free

# Table of Contents

Introduction	0
Features	1
Versioning	2
Install	3
Hi	4
Transport Layer Security	5
Handlers	6
Using Handlers	6.1
Using HandlerFuncs	6.2
Using Annotated	6.3
Using native <code>http.Handler</code>	6.4
Using native <code>http.Handler</code> via <code>iris.ToHandlerFunc()</code>	6.4.1
Middlewares	7
API	8
Declaration	9
Configuration	10
Party	11
Subdomains	12
Named Parameters	13
Static files	14
Send files	15
Render	16
REST	16.1
Templates	16.2
Gzip	17
Streaming	18
Cookies	19

Flash messages	20
Body binder	21
Custom HTTP Errors	22
Context	23
Logger	24
HTTP access control	25
Secure	26
Sessions	27
Websockets	28
Graceful	29
Recovery	30
Plugins	31
Internationalization and Localization	32
Easy Typescript	33
Browser based Editor	34
Routes info	35
Control panel	36
Examples	37



## Table of Contents

- [Introduction](#)
- [Features](#)
- [Versioning](#)
- [Install](#)
- [Hi](#)
- [Transport Layer Security](#)
- [Handlers](#)
  - [Using Handlers](#)
  - [Using HandlerFuncs](#)
  - [Using Annotated](#)
  - [Using native http.Handler](#)
    - [Using native http.Handler via iris.ToHandlerFunc\(\)](#)
- [Middlewares](#)
- [API](#)
- [Declaration](#)

- [Configuration](#)
- [Party](#)
- [Subdomains](#)
- [Named Parameters](#)
- [Static files](#)
- [Send files](#)
- [Render](#)
- [Gzip](#)
- [Streaming](#)
- [Cookies](#)
- [Flash messages](#)
- [Body binder](#)
- [Custom HTTP Errors](#)
- [Context](#)
- [Logger](#)
- [HTTP access control](#)
- [Secure](#)
- [Sessions](#)
- [Websockets](#)
- [Graceful](#)
- [Recovery](#)
- [Plugins](#)
- [Internationalization and Localization](#)
- [Easy Typescript](#)
- [Browser based Editor](#)
- [Routes info](#)
- [Control panel](#)
- [Examples](#)

## Why

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

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

Obviously the `net/http` package is not enough to build well-designed back-end systems for web. But when you realize that, other thoughts are coming to your head:

- Ok the `net/http` package doesn't suits me, but they're so many frameworks, which I have to choose from?!
- Each one of them tells me that it's the best. I don't know what to do!

## The truth

I did a big research and benchmarks with 'wrk' and 'ab' in order to choose which framework suits me and my new project. The results, sadly, were really beaten me, disappointed me.

I was wondering if golang wasn't so fast on the web as I was reading... but, before let Golang and continue to develop with nodejs I told myself:

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

I'm not kidding, these are pretty much the words I told to myself that day [**13 March 2016**].

The same day, later the night, I was reading a book about Greek mythology, there I saw an ancient God's name, inspired immediately and give a name to this new web framework, which was started be written, to **Iris**.

**After two months**, I'm writing this intro.

I'm still here [because Iris has succeed to be the fastest go web framework](#)

# 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 that means. If not, [It's easy to learn about](#)
- **Custom HTTP Errors:** Define your own html templates or plain messages when http errors occurs\*
- **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 in game\*
- **Middlewares:** Create and/or use global or per route middlewares with the Iris' simplicity\*
- **Sessions:** Sessions provides 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 middlewares, render rich content, send file and much more\*
- **Plugins:** You can build your own plugins to inject the Iris framework\*
- **Full API:** All http methods are supported\*
- **Party:** Group routes when sharing the same resources or middlewares. 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:** Besides the fact that Iris has a default main server. You can declare as many as you need\*
- **Zero configuration:** No need to configure anything, unless you're forced to. Default configurations everywhere, which you can change with ease, well structured
- **Zero allocations:** Iris generates zero garbage



# Versioning

Current: **v3.0.0-alpha.2**

Read more about Semantic Versioning 2.0.0

- <http://semver.org/>
- [https://en.wikipedia.org/wiki/Software\\_versioning](https://en.wikipedia.org/wiki/Software_versioning)
- [https://wiki.debian.org/UpstreamGuide#Releases\\_and\\_Versions](https://wiki.debian.org/UpstreamGuide#Releases_and_Versions)

# Install

## Compatible with go1.6+

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

this will update the dependencies also.

If you are connected to the Internet through **China**, according to [this](#) you will be have problem downloading the golang/x/net/context. **Follow the below steps:**

1. <https://github.com/northbright/Notes/blob/master/Golang/china/get-golang-packages-on-golang-org-in-china.md>
2. `$ go get github.com/kataras/iris` **without -u**

# Hi

```
package main

import "github.com/kataras/iris"

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

## The same

```
package main

import "github.com/kataras/iris"

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

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

## Rich Hi with **html/template**

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

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

func main() {
    iris.Config().Render.Template.Engine = config.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

```
// Listen starts the standalone http server
// which listens to the addr parameter which as the form of
// host:port or just port
//
// It panics on error if you need a func to return an error use the
// ex: iris.Listen(":8080")
Listen(addr string)

// ListenWithErr starts the standalone http server
// which listens to the addr parameter which as the form of
// host:port or just 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

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

// ListenTLSWithErr Starts a https server with certificates,
// if you use this method the requests of the form of 'http://' will
// only https:// connections are allowed
// which listens to the addr parameter which as the form of
// host:port or just 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, keyFile string) error
```

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

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





# Handlers

Handlers should implement the Handler interface:

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

## Using Handlers

```
type myHandlerGet struct {  
}  
  
func (m myHandlerGet) Serve(c *iris.Context) {  
    c.Write("From %s", c.PathString())  
}  
  
//and so on  
  
iris.Handle("GET", "/get", myHandlerGet{})  
iris.Handle("POST", "/post", post)  
iris.Handle("PUT", "/put", put)  
iris.Handle("DELETE", "/delete", del)
```

## Using HandlerFuncs

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

```
type HandlerFunc func(*Context)

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

HandlerFuncs should 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 Annotated

Implements the Handler interface

```
///file: userhandler.go
import "github.com/kataras/iris"

type UserHandler struct {
    iris.Handler `get:"/profile/user/:userId"`
}

func (u *UserHandler) Serve(c *iris.Context) {
    userId := c.Param("userId")
    c.Render("user.html", struct{ Message string }{Message: "Hello"})
}
```

```
///file: main.go
iris.Config().Templates.Directory = "templates" // Default is already set
//...register the handler
iris.HandleAnnotated(&UserHandler{})
//...continue writing your wonderful API
```

## 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 *http.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{}))
```

# Middlewares

## Quick view

```
// First point on 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")
```

Middlewares in Iris are not complicated, imagine them as simple Handlers. They should 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.WriteText(403, "Forbidden !!")
    }
}

iris.Use(&myMiddleware{})

iris.Get("/home", func (c *iris.Context){
    c.WriteHTML(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, all before t
})

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

Note that middlewares must come before route declaration.

Make use one of build'n Iris [middlewares](#), view practical [examples here](#)

```
package main

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

type Page struct {
    Title string
}

iris.Config().Templates.Directory = "./yourpath/templates"

iris.Use(logger.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

Let's make a pause,

- Q: Other frameworks needs more lines to start a server, why Iris is different?
- A: Iris gives you the freedom to choose between three ways to declare 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 3rd way, let's take a quick look at the **config.Iris**:

```
// Iris configs for the station
// All fields can be changed before server's listen except the
//
```

```
// MaxRequestBodySize is the only options that can be changed after declaration but before using Config().MaxRequestBodySize = ...
// Render's rest config can be changed after declaration but before using Config().Render.Rest...
// Render's Template config can be changed after declaration but before using Config().Render.Template...
// Sessions config can be changed after declaration but before using Config().Sessions...
// and so on...
Iris struct {
    // MaxRequestBodySize Maximum request body size.
    //
    // The server rejects requests with bodies exceeding this size.
    //
    // By default request body size is unlimited.
    MaxRequestBodySize int
    // PathCorrection corrects and redirects the requested path
    // for example, if /home/ path is requested but no handler
    // then the Router checks if /home handler exists, if yes,
    // (permant)redirects the client to the correct path /home
    //
    // Default is true
    PathCorrection bool

    // Log turn it to false if you want to disable logger,
    // Iris prints/logs ONLY errors, so be careful when you disable it.
    Log bool

    // Profile set to true to enable web pprof (debug profiling)
    // Default is false, enabling makes available these 7 routes
    // /debug/pprof/cmdline
    // /debug/pprof/profile
    // /debug/pprof/symbol
    // /debug/pprof/goroutine
    // /debug/pprof/heap
    // /debug/pprof/threadcreate
    // /debug/pprof/block
    Profile bool

    // ProfilePath change it if you want other url path than the default
```

```

    // Default is /debug/pprof , which means yourhost.com/debug/pprof
    ProfilePath string

    // Sessions the config for sessions
    // contains 3(three) properties
    // Provider: (look /sessions/providers)
    // Secret: cookie's name (string)
    // Life: cookie life (time.Duration)
    Sessions Sessions

    // Render contains the configs for template and rest config
    Render Render
}

```

```

// 3.
package main

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

func main() {
    c := config.Iris{
        Profile:      true,
        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 to more than one Iris station** in the same app, when it's necessary.

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

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

**PathCorrection** corrects and redirects the requested path to the registered path for example, if /home/ path is requested but no handler for this Route found, then the Router checks if /home handler exists, if yes, redirects the client to the correct path /home and VICE - VERSA if /home/ is registered but /home is requested then it redirects to /home/ (Default is true)

- More about configuration [here](#)

# Configuration

Configuration owns the relative package `github.com/kataras/iris/config`

No need to download it separately, it's being downloaded automatically when you installed Iris.

## Why?

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

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

```
irisConfig := config.Iris { Profile: true, PathCorrection: fa  
api := iris.New(irisConfig)
```

- **easy to remember:** `iris` type takes `config.Iris`, sessions takes `config.Sessions`, `iris.Config().Render` is the `config.Render`, `iris.Config().Render.Template` is the `config.Template`, `Logger` takes `config.Logger` and so on...
- **easy to search & find out what features are exists and what you can change:** just navigate to the config folder and open the type you want to learn about, for example `/iris.go` Iris' type configuration is on `/config/iris.go`
- **All structs which receives configuration are already default-setted**, so don't worry too much, but if you ever need 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 that only `config.Iris` has `config.Default()`. Even the plugins have their default configs, to make it easier for you.



- so you can do this **without pre-set a config by 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(templateFromOthers)

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

Below you will find a list of the config structs.

## Search **All Configs**

```
type (
    // Iris configs for the station
    // All fields can be changed before server's listen except the
    //
    // MaxRequestBodySize is the only options that can be changed after
    // using Config().MaxRequestBodySize = ...
    // Render's rest config can be changed after declaration but before
    // using Config().Render.Rest...
    // Render's Template config can be changed after declaration but before
    // using Config().Render.Template...
    // Sessions config can be changed after declaration but before
    // using Config().Sessions...
    // and so on...
    Iris struct {
        // MaxRequestBodySize Maximum request body size.
```

```
//  
// The server rejects requests with bodies exceeding this size  
//  
// By default request body size is unlimited.  
MaxRequestBodySize int  
// PathCorrection corrects and redirects the requested path  
// for example, if /home/ path is requested but no handler  
// then the Router checks if /home handler exists, if yes,  
// (permanently) redirects the client to the correct path /home  
//  
// Default is true  
PathCorrection bool  
  
// Log turn it to false if you want to disable logger,  
// Iris prints/logs ONLY errors, so be careful when you disable  
Log bool  
  
// Profile set to true to enable web pprof (debug profiling)  
// Default is false, enabling makes available these 7 routes  
// /debug/pprof/cmdline  
// /debug/pprof/profile  
// /debug/pprof/symbol  
// /debug/pprof/goroutine  
// /debug/pprof/heap  
// /debug/pprof/threadcreate  
// /debug/pprof/block  
Profile bool  
  
// ProfilePath change it if you want other url path than the default  
// Default is /debug/pprof , which means yourhost.com/debug/pprof/  
ProfilePath string  
  
// Sessions the config for sessions  
// contains 3(three) properties  
// Provider: (look /sessions/providers)  
// Secret: cookie's name (string)  
// Life: cookie life (time.Duration)  
Sessions Sessions  
  
// Render contains the configs for template and rest configuration
```

```

    Render Render
}

// Render struct keeps organise all configuration about rendering
Render struct {
    // Template the configs for template
    Template Template
    // Rest configs for rendering.
    //
    // these options inside this config don't have any relation
    // from github.com/kataras/iris/rest
    Rest Rest
}
)

```

```

type (
    // Rest is a struct for specifying configuration options for the rest
    Rest struct {
        // Appends the given character set to the Content-Type header
        Charset string
        // Gzip enable it if you want to render with gzip compression
        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 nil.
        PrefixJSON []byte
        // Prefixes the XML output with the given bytes.
        PrefixXML []byte
        // Unescape HTML characters "&<>" to their original values.
        UnEscapeHTML bool
        // Streams JSON responses instead of marshalling prior to sending
        StreamingJSON bool
        // Disables automatic rendering of http.StatusInternalServerError
        // Default is false.
        DisableHTTPErrorRendering bool
    }
)

```

```
EngineType uint8
```

```
Template struct {
```

```
    // contains common configs for both HTMLEngine & Pongo as 1
```

```
    Engine      EngineType
```

```
    Gzip        bool
```

```
    IsDevelopment bool
```

```
    Directory   string
```

```
    Extensions  []string
```

```
    ContentType string
```

```
    Charset     string
```

```
    Asset       func(name string) ([]byte, error)
```

```
    AssetNames  func() []string
```

```
    Layout      string
```

```
    HTMLTemplate HTMLTemplate // contains specific configs for
```

```
    Pongo        Pongo // contains specific configs for pong
```

```
}
```

```
HTMLTemplate struct {
```

```
    RequirePartials bool
```

```
    // Delims
```

```
    Left string
```

```
    Right string
```

```
    // Funcs for HTMLTemplate html/template
```

```
    Funcs []template.FuncMap
```

```
}
```

```
Pongo struct {
```

```
    // Filters for pongo2, map[name of the filter] the filter 1
```

```
    // The filters are auto register
```

```
    Filters map[string]pongo2.FilterFunction
```

```
}
```

```
)
```

```
var (
```

```
    universe time.Time // 0001-01-01 00:00:00 +0000 UTC
```

```

    // CookieExpireNever the default cookie's life for sessions, un
    CookieExpireNever = universe
)

const (
    // DefaultCookieName the secret cookie's name for sessions
    DefaultCookieName      = "irissessionid"
    DefaultSessionGcDuration = time.Duration(2) * time.Hour
    // DefaultRedisNetwork the redis network option, "tcp"
    DefaultRedisNetwork = "tcp"
    // DefaultRedisAddr the redis address option, "127.0.0.1:6379"
    DefaultRedisAddr = "127.0.0.1:6379"
    // DefaultRedisIdleTimeout the redis idle timeout option, time
    DefaultRedisIdleTimeout = time.Duration(5) * time.Minute
    // DefaultRedisMaxAgeSeconds the redis storage last parameter (
    DefaultRedisMaxAgeSeconds = 31556926.0 //1 year

)

type (

    // Redis the redis configuration used inside sessions
    Redis struct {
        // Network "tcp"
        Network string
        // Addr "127.0.0.1:6379"
        Addr string
        // Password string .If no password then no 'AUTH'. Default
        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 se

```

```

        // Default 31556926.0 (1 year)
        MaxAgeSeconds int
    }

    // Sessions the configuration for sessions
    // has 4 fields
    // first is the providerName (string) ["memory","redis"]
    // second is the cookieName, the session's name (string) ["mysession"]
    // third is the time which the client's cookie expires
    // forth is the gcDuration (time.Duration)
    // when this time passes it removes the unused sessions from the memory
    Sessions struct {
        // Provider string, usage iris.Config().Provider = "memory"
        // If you want to customize redis then import the package, and use "redis"
        Provider string
        // Cookie string, the session's client cookie name, for example "mysession"
        Cookie string
        //Expires the date which the cookie must expires. Default is 31556926.0
        Expires time.Time
        //GcDuration every how much duration(GcDuration)
        // the memory should be clear for unused cookies (GcDuration)
        //for example: time.Duration(2)*time.Hour.
        // it will check every 2 hours if cookie hasn't be used for 2 hours
        // deletes it from memory until the user comes back,
        // then the session continue to work as it was
        //
        // Default 2 hours
        GcDuration time.Duration
    }
)

```

```

type (
    Logger struct {
        Out      io.Writer
        Prefix   string
        Flag     int
    }
)

```

```

type (
    // Tsconfig the struct for tsconfig.json
    Tsconfig struct {
        CompilerOptions CompilerOptions `json:"compilerOptions"`
        Exclude           []string      `json:"exclude"`
    }

    // CompilerOptions contains all the compiler options used by the compiler
    CompilerOptions struct {
        Declaration          bool   `json:"declaration"`
        Module                string `json:"module"`
        Target                string `json:"target"`
        Watch                bool   `json:"watch"`
        Charset              string `json:"charset"`
        Diagnostics           bool   `json:"diagnostics"`
        EmitBOM               bool   `json:"emitBOM"`
        EmitDecoratorMetadata bool   `json:"emitDecoratorMetadata"`
        ExperimentalDecorators bool   `json:"experimentalDecorators"`
        InlineSourceMap       bool   `json:"inlineSourceMap"`
        InlineSources         bool   `json:"inlineSources"`
        IsolatedModules       bool   `json:"isolatedModules"`
        Jsx                   string `json:"jsx"`
        ReactNamespace        string `json:"reactNamespace"`
        ListFiles             bool   `json:"listFiles"`
        Locale                string `json:"locale"`
        MapRoot               string `json:"mapRoot"`
        ModuleResolution      string `json:"moduleResolution"`
        NewLine                string `json:".newLine"`
        NoEmit                 bool   `json:"noEmit"`
        NoEmitOnError         bool   `json:"noEmitOnError"`
        NoEmitHelpers         bool   `json:"noEmitHelpers"`
        NoImplicitAny         bool   `json:"noImplicitAny"`
        NoLib                  bool   `json:"noLib"`
        NoResolve              bool   `json:"noResolve"`
        SkipDefaultLibCheck    bool   `json:"skipDefaultLibCheck"`
        OutDir                 string `json:"outDir"`
        OutFile                string `json:"outFile"`
        PreserveConstEnums    bool   `json:"preserveConstEnums"`
        Pretty                 bool   `json:"pretty"`
    }

```

```
    RemoveComments      bool    `json:"removeCommer
    RootDir              string  `json:"rootDir"`
    SourceMap            bool    `json:"sourceMap"`
    SourceRoot           string  `json:"sourceRoot"`
    StripInternal        bool    `json:"stripIntern
    SuppressExcessPropertyErrors bool  `json:"suppressExce
    SuppressImplicitAnyIndexErrors bool  `json:"suppressImp
    AllowUnusedLabels    bool    `json:"allowUnusedL
    NoImplicitReturns    bool    `json:"noImplicitRe
    NoFallthroughCasesInSwitch bool  `json:"noFallthroug
    AllowUnreachableCode bool    `json:"allowUnreach
    ForceConsistentCasingInFileNames bool  `json:"forceConsist
    AllowSyntheticDefaultImports bool  `json:"allowSynthet
    AllowJs              bool    `json:"allowJs"`
    NoImplicitUseStrict  bool    `json:"noImplicitUs
  }

  Typescript struct {
    Bin      string
    Dir      string
    Ignore   string
    Tsconfig Tsconfig
    Editor   Editor
  }
)
```



```
var (  
    // DefaultUsername used for default (basic auth)  
    // username in IrisControl's & Editor's default configuration  
    DefaultUsername = "iris"  
    // DefaultPassword used for default (basic auth)  
    // password in IrisControl's & Editor's default configuration  
    DefaultPassword = "admin!123"  
)  
  
// IrisControl the options which iris control needs  
// contains the port (int) and authenticated users with their password  
type IrisControl struct {  
    // Port the port  
    Port int  
    // Users the authenticated users, [username]password  
    Users map[string]string  
}
```

```
type Editor struct {  
    // Host if empty used the iris server's host  
    Host string  
    // Port if 0 4444  
    Port int  
    // WorkingDir if empty "./"  
    WorkingDir string  
    // Username if empty iris  
    Username string  
    // Password if empty admin!123  
    Password string  
}
```

# Party

Let's party with Iris web framework!

```
func main() {

    //log everything middleware

    iris.UseFunc(func(c *iris.Context) {
        println("[Global log] the requested url path is: ", c.Path)
        c.Next()
    })

    // manage all /users
    users := iris.Party("/users", func(c *iris.Context) {
        println("LOG [/users...] This is the middleware for: ", c.Path)
        c.Next()
    })

    {

        users.Post("/login", loginHandler)
        users.Get("/:userId", singleUserHandler)
        users.Delete("/:userId", userAccountRemoveUserHandler)
    }

    // Party inside an existing Party example:

    beta := iris.Party("/beta")

    admin := beta.Party("/admin")
    {
        /// GET: /beta/admin/
        admin.Get("/", func(c *iris.Context){})
        /// POST: /beta/admin/signin
        admin.Post("/signin", func(c *iris.Context){})
        /// GET: /beta/admin/dashboard
    }
}
```

```
    admin.Get("/dashboard", func(c *iris.Context){})  
    /// PUT: /beta/admin/users/add  
    admin.Put("/users/add", func(c *iris.Context){})  
}  
  
iris.Listen(":8080")  
}
```

# Subdomains

Subdomains in Iris are simple [Parties](#).

```
package main

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

func main() {
    // first the subdomains.
    admin := iris.Party("admin.yourhost.com")
    {
        //this will only success on admin.yourhost.com/hey
        admin.Get("/", func(c *iris.Context) {
            c.Write("Welcome to admin.yourhost.com")
        })
        //this will only success on admin.yourhost.com/hey2
        admin.Get("/hey", func(c *iris.Context) {
            c.Write("Hey from admin.yourhost.com")
        })
    }

    iris.Get("/hey", func(c *iris.Context) {
        c.Write("Hey from no-subdomain yourhost.com")
    })

    iris.Listen(":80")
}
```

# 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 "github.com/kataras/iris"

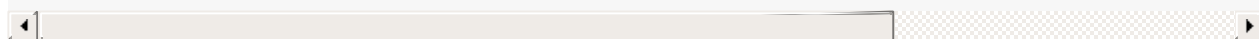
func main() {
    // MATCH to /hello/anywordhere (if PathCorrection:true match a
    // NOT match to /hello or /hello/ or /hello/anywordhere/someth
    iris.Get("/hello/:name", func(c *iris.Context) {
        name := c.Param("name")
        c.Write("Hello %s", name)
    })

    // MATCH to /profile/iris/friends/42
    // (if PathCorrection:true matches also /profile/iris/friends/4
    // NOT match to /profile/ , /profile/something ,
    // NOT match to /profile/something/friends, /profile/something
    // NOT match to /profile/anything/friends/42/something
    iris.Get("/profile/:fullname/friends/:friendId",
        func(c *iris.Context){
            name:= c.Param("fullname")
            //friendId := c.ParamInt("friendId")
            c.WriteHTML(iris.StatusOK, "<b> Hello </b>" + name)
        })

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

## Match anything

```
// Will match any request which url's preffix is "/anything/" and k
iris.Get("/anything/*randomName", func(c *iris.Context) { } )
// Match: /anything/whateverhere/whateveragain , /anything/blablabl
// c.Param("randomName") will be /whateverhere/whateveragain, blabl
// Not Match: /anything , /anything/ , /something
```



# Static files

Serve a static directory

```
// 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 StaticFS
// 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
// 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 exists
// exists and request uri is '/' then display the index.html's content
// 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)
```



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

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

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

## Manual static file serving

Serve static individual file

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

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.PathSep, 1)

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

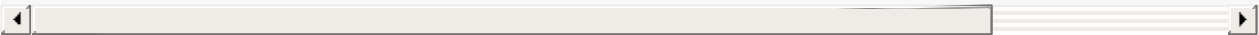
        ctx.ServeFile(path)
    })

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

## Send files

Send a file, force-download to the client

```
// You can define your own "Content-Type" header also, after this 1
// for example: ctx.Response.Header.Set("Content-Type","thecontent,
SendFile(filename string, destinationName string) error
```



```
package main

import "github.com/kataras/iris"

func main() {

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

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

# 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** .

## Templates

Iris gives you the freedom to render templates through [html/template](#) or Django-syntax package [flosch/pongo2](#)

# REST

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

## config.Rest

```
// Appends the given character set to the Content-Type header.  
Charset string  
// Gzip enable it if you want to render with gzip compression.  
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 sending.  
StreamingJSON bool  
// Disables automatic rendering of http.StatusInternalServerError  
// when an error occurs. Default is false.  
DisableHTTPErrorRendering 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,
}

iris.Config().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": "json"})
    })

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

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

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

# Templates

Iris gives you the freedom to render templates through [html/template](#) or Django-syntax package [flosch/pongo2](#) , via **config.Iris.Render.Template = config.Template{}** / **iris.Config().Render.Template = config.Template{}**.

- `HTMLTemplate` is the `html/template`
- `Pongo` is the `flosch/pongo2`

A snippet:

```
iris.Get("/default_standar", func(ctx *iris.Context){
    ctx.Render("index.html", nil) // this will render ./templates/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 {
    // iris.DefaultEngine is the iris.HTMLEngine or iris.PongoEngine
    Engine: config.DefaultEngine
    // Common options for all template engines
    Gzip:          false,
    IsDevelopment: false,
    Directory:     "templates",
    Extensions:    []string{".html"},
    ContentType:   "text/html",
    Charset:       "UTF-8",
    Layout:        "",
```



```
Asset:      nil, // func(name string) ([]byte, error)
AssetNames: nil, // func() []string

// Options when you're using html/template | When Engine == config.HTMLTemplate
HTMLTemplate: config.HTMLTemplate {
    Left: "{{" ,
    Right: "}}",
    Funcs: make([]template.FuncMap, 0),
},

// Option when you're using pongo2 | When Engine == config.Pongo2
Pongo: config.Pongo{Filters: make(map[string]pongo2.FilterFunction),
}

// 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.PongoEngine
iris.Config().Render.Template.Layout = "layout/layout.html" // = "layout/layout.html"
//...

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

```
// HTML builds up the response from the specified template and binds
HTML(status int, name string, binding interface{}, layout ...string) error
// Render same as .HTML but with status to iris.StatusOK (200)
Render(name string, binding interface{}, layout ...string) error
```

## Examples

### HTMLTemplate

```
// main.go

package main

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

type mypage struct {
    Message string
}

func main() {
    iris.Config().Render.Template.Layout = "layouts/layout.html"
    iris.Get("/", func(ctx *iris.Context) {
        if err := ctx.Render("page1.html", mypage{"Message from page1"}); err != nil {
            panic(err)
        }
    })

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

```
<!-- 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">

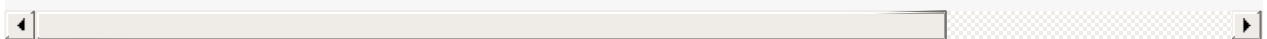
<h1> The message: {{.Message}} </h1>

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

</div>
```

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

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



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

```
// main.go
package main

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

func main() {

    iris.Config().Render.Template.Engine = config.PongoEngine

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

        err := ctx.Render("index.html", map[string]interface{}{"username": "Pongo2"})
        // OR
        //err := ctx.Render("index.html", pongo2.Context{"username": "Pongo2"})

        if err != nil {
            panic(err)
        }
    })

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

```
<!-- 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>
```

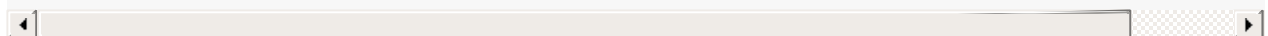
# Gzip

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: gzip'
// 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: gzip'
// header before writing response to w.
//
// WriteGzipLevel doesn't flush response to w for performance reasons.
WriteGzipLevel(w *bufio.Writer, level int) error
```



## How to use

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





# Streaming

Fasthttp has very good support for doing progressive rendering via multiple flushes, streaming. Here is an example, taken from [here](#)

```
package main

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

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

    iris.Listen(":8080")
}

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

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

# 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.

```
// GetFlash get a flash message by it's key
// after this action the messages is removed
// returns string
// if the cookie doesn't exists the string is empty
GetFlash(key string) string

// GetFlashBytes get a flash message by it's key
// after this action the messages is removed
// returns []byte
// and an error if the cookie doesn't exists or decode fails
GetFlashBytes(key string) (value []byte, err error)

// SetFlash sets a flash message
// accepts 2 parameters the key(string) and the value(string)
SetFlash(key string, value string)

// SetFlash sets a flash message
// accepts 2 parameters the key(string) and the value([]byte)
SetFlashBytes(key string, value []byte)
```

### Example

```
package main

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

func main() {

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

    iris.Get("/get", func(c *iris.Context) {
        c.Write("Hello %s", c.GetFlash("name"))
        // the flash message is being deleted after this request done
        // so you can call the c.GetFlash("name")
        // many times without problem
    })

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

        name := c.GetFlash("name")
        if name == "" {
            c.Write("Ok you are coming from /get")
        } else {
            c.Write("Ok you are coming from /set: %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 request's form data
func (ctx *Context) ReadForm(formObject interface{}) error
```

How to use

## JSON

```
package main

import "github.com/kataras/iris"

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

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

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

## XML

```
package main

import "github.com/kataras/iris"

type Company struct {
    Public      bool      `formam:"public"`
    Website     url.URL   `formam:"website"`
    Foundation  time.Time `formam:"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.ReadXML(&Company{}); err != nil {
        panic(err.Error())
    }
}

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

## Form

The form binding came from a fast third party package named [formam](#).

## 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`
- custom types to one of the above types
- a pointer to one of the above types

the nesting in `maps` , `structs` and `slices` can be [ad infinitum](#).

## Custom Marshaling

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

---

## 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 `formam` if the name of a input of form starts lowercase.

```

package main

type InterfaceStruct struct {
    ID    int
    Name  string
}

type Company struct {
    Public      bool      `formam:"public"`
    Website     url.URL   `formam:"website"`
    Foundation  time.Time `formam:"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(iris.StatusText(iris.StatusInternalServerError))
        ctx.SetStatusCode(iris.StatusInternalServerError)
        iris.Logger().Printf("http status: 500 happened!")
    })

    iris.OnError(iris.StatusNotFound, func(ctx *iris.Context) {
        ctx.Write(iris.StatusText(iris.StatusNotFound)) // Outputs:
        ctx.SetStatusCode(iris.StatusNotFound)           // 500

        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.Panic()
    })

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

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

}
```

# Context

- `Write` : `func(string, ...interface{})`
  - `WriteHTML` : `func(int, string)`
  - `Data` : `func(status int, v []byte) error`
  - `HTML` : `func(status int, name string, binding interface{}, htmlOpt ...invalid type) error`
  - `Render` : `func(name string, binding interface{}, htmlOpt ...invalid type) error`
  - `JSON` : `func(status int, v interface{}) error`
  - `JSONP` : `func(status int, callback string, v interface{}) error`
  - `Text` : `func(status int, v string) error`
  - `XML` : `func(status int, v interface{}) error`
  - `ExecuteTemplate` : `func(*html/template.Template, interface{}) error`
  - `ServeContent` : `func(io.ReadSeeker, string, time.Time) error`
  - `ServeFile` : `func(string) error`
  - `SendFile` : `func(filename string, destinationName string) error`
  - `Stream` : `func(func(*bufio.Writer))`
- 
- `Get` : `func(interface{}) interface{}`
  - `GetString` : `func(interface{}) string`
  - `GetInt` : `func(interface{}) int`
  - `Set` : `func(interface{}, interface{})`
  - `SetCookie` : `func(*invalid type)`
  - `SetCookieKV` : `func(string, string)`
  - `RemoveCookie` : `func(string)`
  - `GetFlash` : `func(string) string`
  - `GetFlashBytes` : `func(string) ([]byte, error)`
  - `SetFlash` : `func(string, string)`
  - `SetFlashBytes` : `func(string, []byte)`
- 
- `SetContentType` : `func([]string)`
  - `SetHeader` : `func(string, []string)`
  - `Redirect` : `func(string, ...int)`
  - `NotFound` : `func()`
  - `Panic` : `func()`
  - `EmitError` : `func(int)`
- 
- `Param` : `func(string) string`
  - `ParamInt` : `func(string) (int, error)`
  - `URLParam` : `func(string) string`
  - `URLParamInt` : `func(string) (int, error)`
  - `URLParams` : `func() map[string][]string`
  - `MethodString` : `func() string`
  - `HostString` : `func() string`
  - `PathString` : `func() string`
  - `RequestIP` : `func() string`
  - `RemoteAddr` : `func() string`
  - `RequestHeader` : `func(k string) string`
  - `PostFormValue` : `func(string) string`

- ReadJSON : func(interface{}) error
- ReadXML : func(interface{}) error
- ReadForm : func(formObject interface{}) error
  
- Deadline : func() (deadline time.Time, ok bool)
- Done : func() <-chan struct{}
- Err : func() error
- Value : func(key interface{}) interface{}
- Reset : func(reqCtx \*invalid type)
- Clone : func() \*Context
- Do : func()
- Next : func()
- StopExecution : func()
- IsStopped : func() bool
- GetHandlerName : func() string

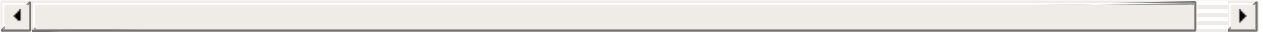
Inside the [examples](#) you will find practical code

# Logger

[This is a middleware](#)

Logs the incoming requests

```
Custom(writer io.Writer, prefix string, flag int) iris.HandlerFunc  
Default() iris.HandlerFunc
```



How to use



```
package main

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

func main() {

    iris.UseFunc(logger.Default())
    // iris.UseFunc(logger.New(config.DefaultLogger()))

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

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

    iris.Get("/3", func(ctx *iris.Context) {
        ctx.Write("hello")
    })

    iris.Listen(":80")
}
```

# 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 request can be made from
// If the special "*" value is present in the list, all origins are allowed
// An origin may contain a wildcard (*) to replace 0 or more characters
// (i.e.: http://*.domain.com). Usage of wildcards implies a strict origin policy
// Only one wildcard can be used per origin.
// Default value is ["*"]
AllowedOrigins []string

// AllowOriginFunc is a custom function to validate the origin.
// as argument and returns true if allowed or false otherwise.
// set, the content of AllowedOrigins is ignored.
AllowOriginFunc func(origin string) bool

// AllowedMethods is a list of methods the client is allowed to use for
// cross-domain requests. Default value is simple methods (GET, POST)
AllowedMethods []string

// AllowedHeaders is list of non simple headers the client is allowed to use
// cross-domain requests.
// If the special "*" value is present in the list, all headers are allowed.
// Default value is [] but "Origin" is always appended to the list.
AllowedHeaders []string

// AllowedHeadersAll bool
// If true, all headers are allowed.

// ExposedHeaders indicates which headers are safe to expose to a
// API specification
ExposedHeaders []string

// AllowCredentials indicates whether the request can include user
// cookies, HTTP authentication or client side SSL certificates
AllowCredentials bool

// MaxAge indicates how long (in seconds) the results of a preflight
// can be cached
MaxAge int

// OptionsPassthrough instructs preflight to let other potential
// process the OPTIONS method. Turn this on if your application
// OptionsPassthrough bool

// Debugging flag adds additional output to debug server side
// Debug bool
```

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

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

## Example

```
package main

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

# Secure

This is a middleware

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

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

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

## Example

```
package main

import (
    "github.com/kataras/iris"
    "github.com/kataras/iris/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-Security 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 header.
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 HTTPS.
//If you want to force it to always be added, set to true.
//`IsDevelopment` still overrides this. Default is false.
FrameDeny:            true,
// If FrameDeny is set to true, adds the X-Frame-Options header
//the value of `DENY`. Default is false.
CustomFrameOptionsValue: "SAMEORIGIN",
// CustomFrameOptionsValue allows the X-Frame-Options header
//value to be set with
//a custom value. This overrides the FrameDeny option.
ContentTypeNosniff:   true,
// If ContentTypeNosniff is true, adds the X-Content-Type-Options
//header with the value `nosniff`. Default is false.
BrowserXSSFilter:     true,
// If BrowserXssFilter is true, adds the X-XSS-Protection header
//with the value `1;mode=block`. Default is false.
ContentSecurityPolicy: "default-src 'self'",
```

```
// ContentSecurityPolicy allows the Content-Security-Policy
//header value to be set with a custom value. Default is ""
PublicKey: `pin-sha256="base64+primary=="`; p
// 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 false
}))

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-locate 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 info
    //iris.Config().Sessions.Expires = time.Time....
    // GcDuration every how much duration(GcDuration) the memory should
    //iris.Config().Sessions.GcDuration = time.Duration(2) *time.Hour

    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().GetString("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")
}
```

### 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().GetString("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")
}
```

Example customized **config.Redis**

```
// Redis the redis configuration used inside sessions
Redis struct {
    // Network "tcp"
    Network string
    // Addr "127.0.0.1:6379"
    Addr string
    // Password string .If no password then no 'AUTH'. Default
    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
}
```

```
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().GetString("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) ["mysessionid"]
// 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
// this is for the client's/browser's Cookie life time(expires) also

New(provider string, cName string, gcDuration time.Duration) *sessions.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 session is stored in memory
        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("name"))
    })

    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{}]interface{}
// session.VisitAll(func(key interface{}, value interface{}) { /* ... */ })
}
```

Example **redis** with config.Redis defaults

The default redis client points to 127.0.0.1:6379



```
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'. Default
    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
}
```

```
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 cli
    // 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 setting 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](#)

---

How to use

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

func chat(c *websocket.Conn) {
    // defer c.Close()
    // mt, message, err := c.ReadMessage()
    // c.WriteMessage(mt, message)
}

var upgrader = websocket.New(chat) // use default options
//var upgrader = websocket.Custom(chat, 1024, 1024) // customized c
// var upgrader = websocket.New(chat).DontCheckOrigin() // it's use

func myChatHandler(ctx *iris.Context) {
    err := upgrader.Upgrade(ctx) // returns only error, executes the
}

func main() {
    iris.Get("/chat_back", myChatHandler)
    iris.Listen(":80")
}
```

The iris/websocket package has been converted from the gorilla/websocket. If you want to see more examples just go [here](#) and make the conversions as you see in 'How to use' before.

# Graceful

This is a package

Enables graceful shutdown.

```
package main

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

```
package main

import (
    "github.com/kataras/iris"
    "github.com/kataras/iris/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 only the requests. Middleware starts it's actions after the server listen, Plugin on the other hand starts working when you registered them, from the begin, to the end. Look how it's interface looks:

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

// IPluginGetDescription implements the GetDescription() string
IPluginGetDescription interface {
    // GetDescription has to returns the description of what th
    GetDescription() string
}

// IPluginGetDescription implements the Activate(IPluginContain
IPluginActivate interface {
    // Activate called BEFORE the plugin being added to the plu
    // if Activate returns none nil error then the plugin is no
    // it is being called only one time
    //
    // PluginContainer parameter used to add other plugins if t
    Activate(IPluginContainer) error
}

// IPluginPreHandle implements the PreHandle(IRoute) method
IPluginPreHandle interface {
    // PreHandle it's being called every time BEFORE a Route is
    //
    // parameter is the Route
    PreHandle(IRoute)
```

```
}
// IPluginPostHandle implements the PostHandle(IRoute) method
IPluginPostHandle interface {
    // PostHandle it's being called every time AFTER a Route is
    //
    // parameter is the Route
    PostHandle(IRoute)
}
// IPluginPreListen implements the PreListen(*Station) method
IPluginPreListen interface {
    // PreListen it's being called only one time, BEFORE the Server
    // is used to do work at the time all other things are ready
    // parameter is the station
    PreListen(*Station)
}
// IPluginPostListen implements the PostListen(*Station) method
IPluginPostListen interface {
    // PostListen it's being called only one time, AFTER the Server
    // parameter is the station
    PostListen(*Station)
}
// IPluginPreClose implements the PreClose(*Station) method
IPluginPreClose interface {
    // PreClose it's being called only one time, BEFORE the Iris
    // any plugin cleanup/clear memory happens here
    //
    // The plugin is deactivated after this state
    PreClose(*Station)
}
```

A small example, imagine that you want to get all routes registered to your server (OR modify them at runtime), with their time registered, methods, (sub)domain and the path, what would you do on other frameworks when you want something from the framework which it doesn't support out of the box? and what you can do with Iris:

```
//file myplugin.go
package main
```

```
import (  
    "time"  
  
    "github.com/kataras/iris"  
)  
  
type RouteInfo struct {  
    Method      string  
    Domain      string  
    Path        string  
    TimeRegistered time.Time  
}  
  
type myPlugin struct {  
    routes []RouteInfo  
}  
  
func NewMyPlugin() *myPlugin {  
    return &myPlugin{routes: make([]RouteInfo, 0)}  
}  
  
//  
// Implement our plugin, you can view your inject points - listener  
//  
// Implement the PostHandle, because this is what we need now, we r  
func (i *myPlugin) PostHandle(route iris.IRoute) {  
    myRouteInfo := &RouteInfo{}  
    myRouteInfo.Method = route.GetMethod()  
    myRouteInfo.Domain = route.GetDomain()  
    myRouteInfo.Path = route.GetPath()  
  
    myRouteInfo.TimeRegistered = time.Now()  
  
    i.routes = append(i.routes, myRouteInfo)  
}  
  
// PostListen called after the server is started, here you can do a  
// you have the right to access the whole iris' Station also, here  
// for example let's print to the server's stdout the routes we col
```

```
func (i *myPlugin) PostListen(s *iris.Station) {
    s.Logger.Printf("From MyPlugin: You have registered %d routes ",
        //do what ever you want, you have imagination do more than this
    )
}

//
```

Let's register our plugin:

```
//file main.go
package main

import "github.com/kataras/iris"

func main() {
    iris.Plugins().Add(NewMyPlugin())
    //the plugin is running and saves all these routes
    iris.Get("/", func(c *iris.Context){})
    iris.Post("/login", func(c *iris.Context){})
    iris.Get("/login", func(c *iris.Context){})
    iris.Get("/something", func(c *iris.Context){})

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

Output:

```
| From MyPlugin: You have registered 4 routes
```

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](#)

# 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/kataras/iris/middleware/i18n"
)

func main() {

    iris.Use(i18n.I18nHandler(i18n.Options{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"}}))
    // or iris.UseFunc(i18n.I18n(...))
    // or iris.Get("/", i18n.I18n(...), func (ctx *iris.Context) {

    iris.Get("/", func(ctx *iris.Context) {
        hi := ctx.GetFmt("translate")("hi", "maki") // hi is the
        language := ctx.Get("language") // language is the language
        ctx.Write("From the language %s translated output: %s",
        hi, language)
    })

    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. Default is nil

All these are optional



## How to use

```
package main

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

func main(){
    ts := config.Typescript {
        Dir: "./scripts/src",
        Tsconfig: config.Tsconfig{Module: "commonjs", Target: "es5"
    }
    // or config.DefaultTypescript()

    iris.Plugins().Add(typescript.New(ts)) //or with the default op

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

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

Enable [web browser editor](#)

```
ts := config.Typescript {
    //...
    Editor: config.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/kataras/iris/config"
    "github.com/kataras/iris/plugin/editor"
)

func main(){
    e := editor.New()
    // config.Editor{ Username: "admin", Password: "admin!123", Port
    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(...)`

# Routes information

[This is a plugin](#)

Collects & stores all registered routes.

```
type RouteInfo struct {  
    Method      string  
    Domain      string  
    Path        string  
    RegisteredAt time.Time  
}
```

## Example

```
package main  
  
import (  
    "github.com/kataras/iris"  
    "github.com/kataras/iris/plugin/routesinfo"  
)  
  
func main() {  
  
    info := routesinfo.New()  
    iris.Plugins().Add(info)  
  
    iris.Get("/yourpath", func(c *iris.Context) {  
        c.Write("yourpath")  
    })  
  
    iris.Post("/otherpostpath", func(c *iris.Context) {  
        c.Write("other post path")  
    })  
  
    all := info.All()  
    // allget := info.ByMethod("GET") -> slice
```

```
// alllocalhost := info.ByDomain("localhost") -> slice
// bypath:= info.ByPath("/yourpath") -> slice
// bydomainandmethod:= info.ByDomainAndMethod("localhost","GET")
// bymethodandpath:= info.ByMethodAndPath("GET","/yourpath") -> slice
//single (it could be slice for all domains too but it's not)

println("The first registered route was: ", all[0].Path, "registered")
println("All routes info:")
for i:= range all {
    println(all[i].String())
    //outputs->
    // Domain: localhost Method: GET Path: /yourpath RegisteredAt: 2017-01-01 00:00:00
    // Domain: localhost Method: POST Path: /otherpostpath RegisteredAt: 2017-01-01 00:00:00
}
iris.Listen(":8080")
}
```

# Control panel

This is a [plugin](#) which is working but not finished.

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.Web(port int, authenticatedUsers map[string]string) ir:
```

## Example

```
package main

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

func main() {

    iris.Plugins().Add(iriscontrol.Web(9090, map[string]string{
        "irisusername1": "irispassword1",
        "irisusername2": "irispassowrd2",
    }))
    //or
    // import "github.com/kataras/iris/config"
    // ....
    // iriscontrol.New(config.IrisControl{...})

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

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

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