

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
Party	10
Subdomains	11
Named Parameters	12
Static files	13
Send files	14
Render	15
Gzip	16
Streaming	17
Cookies	18
Flash messages	19
Body binder	20
Custom HTTP Errors	21

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

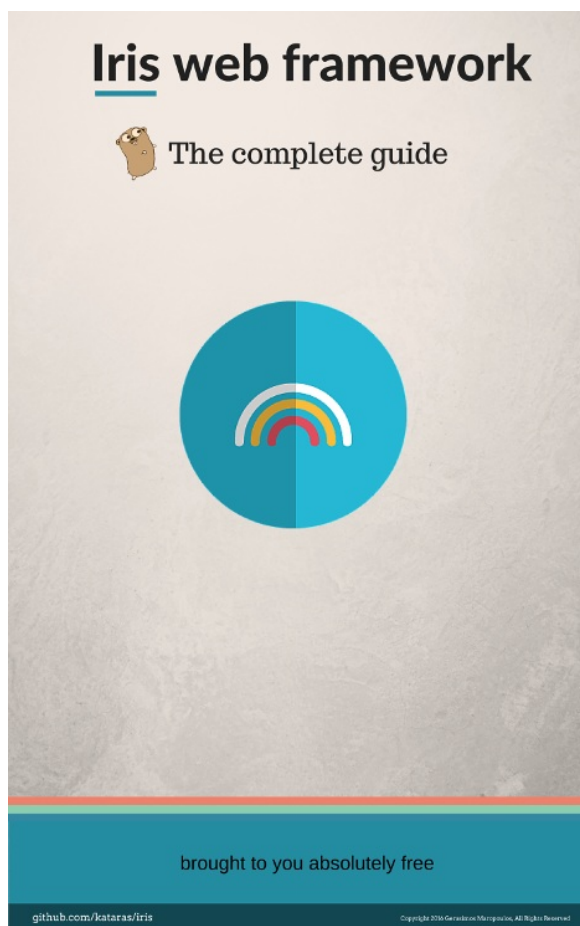


Table of Contents

- [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](#)

- [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

- **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 allocations:** Iris generates zero garbage

Versioning

Current: **v2.2.3**

Read more about Semantic Versioning 2.0.0

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

Install

Compatible with go1.6+

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

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:**

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**

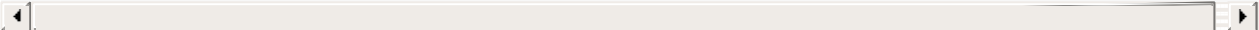
Hi

The name of this framework came from **Greek mythology**, **Iris** was the name of the Greek goddess of the **rainbow**.

```
package main

import "github.com/kataras/iris"

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



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", struct{ Message string }{Message: "Hello User"})
}
```

```
///file: main.go
//...cache the html files, if you the content of any html file char
iris.Config().Render.Directory = "./templates"
//...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().Render.Directory = "./yourpath/templates"

iris.Use(logger.Logger())

iris.Get("/", func(c *iris.Context) {
    c.Render("index", 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:
iris.New(iris.IrisConfig{...})

```
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 **iris.IrisConfig**:

```
IrisConfig 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,
    // and VICE - VERSA if /home/ is registered but /home is requested.
    //
    // 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
    // Render specify configs for rendering
    Render iris.RenderConfig
}
```

```
// 3.
func thirdMethod() {

    config := &IrisConfig{
        PathCorrection:      true,
        MaxRequestBodySize: -1,
        Log:                  true,
        Profile:              false,
        ProfilePath:         DefaultProfilePath,
        Render: &RenderConfig{
            Directory:      "templates",
            Asset:           nil,
            AssetNames:     nil,
            Layout:          "",
            Extensions:     []string{".html"},
            Funcs:            []template.FuncMap{},
            Delims:           Delims{"{{", "}}"},
            Charset:          DefaultCharset,
            IndentJSON:        false,
            IndentXML:         false,
            PrefixJSON:        []byte(""),
            PrefixXML:         []byte(""),
            HTMLContentType:   "text/html",
            IsDevelopment:     false,
            UnEscapeHTML:      false,
            StreamingJSON:      false,
            RequirePartials:   false,
            DisableHTTPErrorsRendering: false,
        }}//these are the default values that you can change
    // DefaultProfilePath = "/debug/pprof"
    // DefaultCharset = "UTF-8"

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

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

As you can see there are some options that you can change at your iris declaration.

For example if we do that...

```
package main

import "github.com/kataras/iris"

func main() {
    config := iris.IrisConfig{
        Profile:      true,
        ProfilePath:   "/mypath/debug",
    }

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

run it, then you can open your browser, type

'localhost:8080/mypath/debug/profile' at the location input field and you should see a webpage shows you informations about CPU.

For profiling & debug there are seven (7) generated pages ('/debug/pprof/' is the default profile path, which on previous example we changed it to '/mypath/debug'):

1. /debug/pprof/cmdline
2. /debug/pprof/profile
3. /debug/pprof/symbol
4. /debug/pprof/goroutine
5. /debug/pprof/heap
6. /debug/pprof/threadcreate
7. /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)

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"

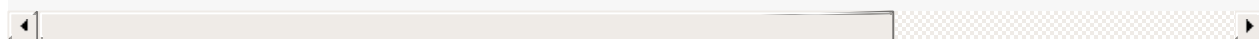
func main() {
    // MATCH to /hello/anywordhere (if PathCorrection:true match a
    // NOT match to /hello or /hello/ or /hello/anywordhere/someth
    iris.Get("/hello/:name", func(c *iris.Context) {
        name := c.Param("name")
        c.Write("Hello %s", name)
    })

    // MATCH to /profile/iris/friends/42 (if PathCorrection:true r
    // 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 and request url
// 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

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

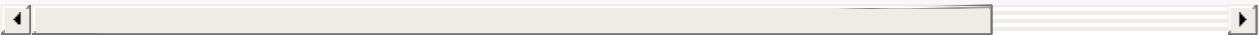
        ctx.ServeFile(path)
    })

    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

This is a package

Provides functionality for easily, one line, rendering JSON, XML, text, binary data, and HTML templates.

All functions are inside Context, options declaration at the configuration state.

Usage

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

- HTML/Render: Uses the [html/template](#) package to render HTML templates.
- 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`.

```
// main.go
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.Get("/html", func(ctx *iris.Context) {
        // Assumes you have a template in ./templates called "example.html"
        // $ mkdir -p templates && echo "<h1>Hello HTML world.</h1>" > templates/example.html
        ctx.HTML(iris.StatusOK, "example", nil)
    })

    // ctx.Render is the same as ctx.HTML but with default 200 status code
    iris.Get("/html2", func(ctx *iris.Context) {
        // Assumes you have a template in ./templates called "example.html"
        // $ mkdir -p templates && echo "<h1>Hello HTML world.</h1>" > templates/example.html
        ctx.Render("example", nil)
    })

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

```
<!-- templates/example.html -->
<h1>Hello {{.}}.</h1>
```

Available Options

Render comes with a variety of configuration options (*Note: these are not the default option values. See the defaults below.*):

```
// ...
renderOptions := &iris.RenderConfig{
    Directory: "templates", // Specify what path to load the templates from
    Asset: func(name string) ([]byte, error) { // Load from an Asset
        return []byte("template content"), nil
    },
    AssetNames: func() []string { // Return a list of asset names to load
        return []string{"filename.html"}
    },
    Layout: "layout", // Specify a layout template. Layouts can call the Asset function
    Extensions: []string{".tmpl", ".html"}, // Specify extensions to load
    Funcs: []template.FuncMap{AppHelpers}, // Specify helper functions
    Delims: iris.Delims{"{{", "}}"}, // Sets delimiters to the specified ones
    Charset: "UTF-8", // Sets encoding for json and html content-type
    Gzip: false, // Enable it if you want to render using gzip compression
    IndentJSON: true, // Output human readable JSON.
    IndentXML: true, // Output human readable XML.
    PrefixJSON: []byte("{}'\n"), // Prefixes JSON responses with {}'\n
    PrefixXML: []byte("<?xml version='1.0' encoding='UTF-8'?>"), // Prefixes XML responses with <?xml version='1.0' encoding='UTF-8'?>
    HTMLContentType: "application/xhtml+xml", // Output XHTML content-type
    IsDevelopment: true, // Render will now recompile the templates on every request
    UnEscapeHTML: true, // Replace ensure '&<>' are output correctly
    StreamingJSON: true, // Streams the JSON response via json.Encoder
    RequirePartials: true, // Return an error if a template is missing
    DisableHTTPErrorRendering: true, // Disables automatic rendering of HTTP errors
})
// ...
```

Default Options

These are the preset options for Render:

```
// Is the same as the default configuration options:

renderOptions = &iris.RenderConfig{
    Directory: "templates",
    Asset: nil,
    AssetNames: nil,
    Layout: "",
    Extensions: []string{".html"},
    Funcs: []template.FuncMap{},
    Delims: iris.Delims{"{{", "}}"},
    Charset: "UTF-8",
    Gzip: false,
    IndentJSON: false,
    IndentXML: false,
    PrefixJSON: []byte(""),
    PrefixXML: []byte(""),
    HTMLContentType: "text/html",
    IsDevelopment: false,
    UnEscapeHTML: false,
    StreamingJSON: false,
    RequirePartials: false,
    DisableHTTPErrorRendering: false,
})
```

JSON vs Streaming JSON

By default, Render does **not** stream JSON to the `iris.Context.Response`. It instead marshalls your object into a byte array, and if no errors occurred, writes that byte array to the `iris.Context.Response`. This is ideal as you can catch errors before sending any data.

If however you have the need to stream your JSON response (ie: dealing with massive objects), you can set the `StreamingJSON` option to true. This will use the `json.Encoder` to stream the output to the `iris.Context.Response`. If an

error occurs, you will receive the error in your code, but the response will have already been sent. Also note that streaming is only implemented in `render.JSON` and not `render.JSONP`, and the `UnEscapeHTML` and `Indent` options are ignored when streaming.

Loading Templates

By default Render will attempt to load templates with a `.html` extension from the `"templates"` directory. Templates are found by traversing the templates directory and are named by path and basename. For instance, the following directory structure:

```
templates/  
|  
|__ admin/  
|    |  
|    |__ index.html  
|    |  
|    |__ edit.html  
|  
|__ home.html
```

Will provide the following templates:

```
admin/index  
admin/edit  
home
```

You can also load templates from memory by providing the `Asset` and `AssetNames` options, e.g. when generating an asset file using [go-bindata](#).

Layouts

Render provides `yield` and `partial` functions for layouts to access:

```
// ...

renderOptions := &iris.RenderConfig{
    Layout: "layout",
    Gzip: true,
}

iris.SetRenderConfig(renderOptions)
// or api := iris.New(Render: renderOptions)

// ...
```

```
<!-- templates/layout.html -->
<html>
  <head>
    <title>My Layout</title>
    <!-- Render the partial template called `css-$current_template` -->
    {{ partial "css" }}
  </head>
  <body>
    <!-- render the partial template called `header-$current_template` -->
    {{ partial "header" }}
    <!-- Render the current template here -->
    {{ yield }}
    <!-- render the partial template called `footer-$current_template` -->
    {{ partial "footer" }}
  </body>
</html>
```

`current` can also be called to get the current template being rendered.

```
<!-- templates/layout.html -->
<html>
  <head>
    <title>My Layout</title>
  </head>
  <body>
    This is the {{ current }} page.
  </body>
</html>
```

Partials are defined by individual templates as seen below. The partial template's name needs to be defined as "{partial name}-{template name}".

```
<!-- templates/home.html -->
{{ define "header-home" }}
<h1>Home</h1>
{{ end }}

{{ define "footer-home" }}
<p>The End</p>
{{ end }}
```

By default, the template is not required to define all partials referenced in the layout. If you want an error to be returned when a template does not define a partial, set `RenderConfig.RequirePartials = true`.

Character Encodings

Render will automatically set the proper Content-Type header based on which function you call.

In order to change the charset, you can set the `Charset` within the

`RenderConfig` to your encoding value, or `Iris.DefaultCharset = "UTF-8"`

```
// main.go
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.DefaultCharset = "ISO-8859-1"
    // or iris.SetRenderConfig(&iris.RenderConfig{ Charset: "ISO-8859-1" })

    //...
}
```

Error Handling

The rendering functions return any errors from the rendering engine. By default, they will also write the error to the HTTP response and set the status code to 500. You can disable this behavior so that you can handle errors yourself by setting

```
RenderConfig.DisableHTTPErrorRendering: true .
```

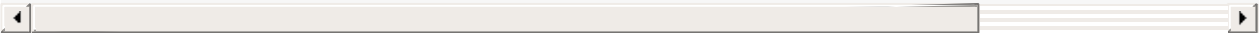
```
renderOptions := &iris.RenderConfig{
    DisableHTTPErrorRendering: true,
}

iris.SetRenderConfig(renderOptions)

//...
func (ctx *iris.Context) {
    err := ctx.HTML(iris.StatusOK "example", "World")
    if err != nil{
        ctx.Redirect("/my-custom-500", iris.StatusFound)
    }
}
```

Templates

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



Example


```
//  
// FILE: ./main.go  
//  
package main  
  
import (  
    "github.com/kataras/iris"  
)  
  
type mypage struct {  
    Title    string  
    Message string  
}  
  
func main() {  
  
    //optionally - before the load.  
    //iris.Config().Render.Delims = iris.Delims{Left:"${", Right: "  
    //iris.Config().Render.Funcs = template.FuncMap(...)  
  
    iris.Config().Render.Directory = "templates" // Default "templa  
    iris.Config().Render.Layout = "layout" // Default is ""  
    iris.Config().Render.Gzip = true           // Default is false  
    //...  
  
    iris.Get("/", func(ctx *iris.Context) {  
        ctx.Render("mypage", mypage{"My Page title", "Hello world!"  
    })  
  
    println("Server is running at :8080")  
    iris.Listen(":8080")  
}
```

```
<!--  
  FILE: ./templates/layout.html  
-->  
<html>  
  <head>  
    <title>My Layout</title>  
  
  </head>  
  <body>  
    <!-- Render the current template here -->  
    {{ yield }}  
  </body>  
</html>
```

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

Gzip

Gzip compression is easy.

For **auto-gzip** all responses, look the Gzip option at the iris.RenderConfig [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 comming from /get")
        } else {
            c.Write("Ok you are comming 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`

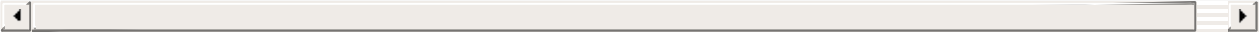
Inside the [examples](#) you will find practical code

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.Custom(writer io.Writer, prefix string,

    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 make cross-domain
// requests. Default value is simple methods (GET, POST, HEAD).
AllowedMethods []string

// AllowedHeaders is list of non simple headers the client is allowed to make
// 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 requesting
// API specification
ExposedHeaders []string

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

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

// OptionsPassthrough instructs preflight to let other potential clients
// process the OPTIONS method. Turn this on if your application uses a
// reverse proxy that can handle the OPTIONS method.
OptionsPassthrough bool

// Debugging flag adds additional output to debug server side CORS logic
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.

```
// 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) ["mysession"]
// 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) *session
```

Example **memory**

```
package main

import (
    "time"

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

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

var sess *sessions.Manager

func init() {
    sess = sessions.New("memory", "irissessionid", time.Duration(60))
}

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 default configuration

The default redis client points to 127.0.0.1:6379

```
package main

import (
    "time"

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

    _ "github.com/kataras/iris/sessions/providers/redis"
    // here we add the redis provider and store
    //with the default redis client points to 127.0.0.1:6379
)

var sess *sessions.Manager

func init() {
    sess = sessions.New("redis", "irissessionid", time.Duration(60))
}

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

Example **redis** with custom configuration

```
type Config 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 5 * time.Minute  
    IdleTimeout time.Duration  
    //Prefix "myprefix-for-this-website". Default ""  
    Prefix string  
    // MaxAgeSeconds how much long the redis should keep the session  
    MaxAgeSeconds int  
}
```

```
package main

import (
    "time"

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

    "github.com/kataras/iris/sessions/providers/redis"
    // here we add the redis provider and store
    //with the default redis client points to 127.0.0.1:6379
)

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.Redis.Config.Network = "tcp"
    redis.Redis.Config.Addr = "127.0.0.1:6379"
    redis.Redis.Config.Prefix = "myprefix-for-this-website"

    sess = sessions.New("redis", "irissessionid", time.Duration(60))
}

//...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, we should 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 := "astaxie%^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 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/websocket"
    "github.com/kataras/iris"
)

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

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**: `&typescript.Tsconfig{}`, here you can set all compilerOptions if no tsconfig.json exists inside the 'Dir'
- **Editor**: `typescript.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/plugin/typescript"
)

func main(){
    ts := typescript.Options {
        Dir: "./scripts/src",
        Tsconfig: &typescript.Tsconfig{Module: "commonjs", Target:
    }
    // or typescript.DefaultTsconfig()

    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 := typescript.Options {
    //...
    Editor: typescript.Editor("username", "passowrd")
    //...
}
```

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/plugin/editor"
)

func main(){
    e := editor.New("username", "password").Port(4444).
        Dir("/path/to/the/client/side/directory")

    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") -> slice
// 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: 2016-01-01 00:00:00
    // Domain: localhost Method: POST Path: /otherpostpath RegisteredAt: 2016-01-01 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": "irispasword1",
        "irisusername2": "irispasowrd2",
    }))

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

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

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