

Table of Contents

Home	1.1
About	1.2
Benchmarks	1.3
Support	1.4
Installing Iris	1.5
Getting Started	1.6
Host	1.7
Automatic Public Domain with TLS	1.7.1
Configuration	1.8
Routing	1.9
Path Parameter Types	1.9.1
Reverse Lookups	1.9.2
Middleware	1.9.3
Handle HTTP errors	1.9.4
Subdomains	1.9.5
Wrap the Router	1.9.6
Override Context	1.9.7
Context Methods	1.9.8
API Versioning	1.10
Content Negotiation	1.11
Response Recorder	1.12
HTTP Referer	1.13
Request Authentication	1.14
URL Query Parameters	1.15
Forms	1.16
Model Validation	1.17
Cache	1.18
File Server	1.19
View	1.20
Cookies	1.21
Sessions	1.22
Database	1.22.1
Flash Messages	1.22.2
Websockets	1.23
Dependency Injection	1.24
MVC	1.25
Testing	1.26
Examples	1.27
Starter Kits	1.28
Publications	1.29

Home

Welcome to **Iris** - the fastest backend web framework for [Go](#).

This book is the main source of documentation for **teams and individuals developers** working with (or contributing to) the Iris project.

Table of Contents

- [About](#)
- [Benchmarks](#)
- [Support](#)
- [Installing Iris](#)
- [Getting Started](#)
- [Host](#)
 - [Automatic Public Domain with TLS](#)
- [Configuration](#)
- [Routing](#)
 - [Path Parameter Types](#)
 - [Reverse Lookups](#)
 - [Middleware](#)
 - [Handle HTTP errors](#)
 - [Subdomains](#)
 - [Wrap the Router](#)
 - [Override Context](#)
 - [Context Methods](#)
- [API Versioning](#)
- [Content Negotiation](#)
- [Response Recorder](#)
- [HTTP Referer](#)
- [Request Authentication](#)
- [URL Query Parameters](#)
- [Forms](#)
- [Model Validation](#)
- [Cache](#)
- [File Server](#)
- [View](#)
- [Cookies](#)
- [Sessions](#)
 - [Database](#)
 - [Flash Messages](#)
- [Websockets](#)
- [Dependency Injection](#)
- [MVC](#)
- [Testing](#)
- [Examples](#)
- [Starter Kits](#)
- [Publications](#)

About

Iris is a fast, simple yet fully featured and very efficient web framework written in [Go](#).

Iris provides a beautifully expressive and easy to use foundation for your next website or API.

Iris offers a complete and decent support to its users.

Our Philosophy

The Iris philosophy is to provide robust tooling for HTTP, making it a great solution for single page applications, web sites, hybrids, or public HTTP APIs. Keep note that, so far, iris is the fastest web framework ever created in terms of real-world performance.

Iris does not force you to use any specific ORM or template engine. With support for the most powerful and fast template engines, you can quickly craft the perfect application.

Why Yet Another Web Framework?

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

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

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

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

The truth

I did some deep research and benchmarks with [wrk](#) and [ab](#) in order to choose which framework suits me and my new project. The results, unfortunately, were really disappointing to me.

I started wondering if Golang wasn't as fast for `net` or `http` applications as I was reading from different kind of online sources... but right before I leave Golang for good and continue to develop with `nodejs` and `.net` core, I whispered to myself:

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

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

The same day, later the night, I was reading a book about Greek mythology. I saw an ancient goddess name there and I was immediately inspired to give her name to this new web framework I was already started to design and code - **Iris**.

Some weeks later, Iris was trending on position #1 in Github for **all languages**, this is very rare phenomenon especially for a new personal project, back then, with such a young developer behind it. After that, as was reasonable, competitors who couldn't handle their losses, and individuals began to pull out themselves and their jealousy about the success of others, started to randomly throw defamations and fake news around. Of course they never even thought to argument on the fact that Iris the fastest and greatest in terms of performance and features because numbers never lie and their whole narrative would fall very fast. Instead they tried to bullying me through my personality, they tried to put me down in order to stop the active development of Iris.

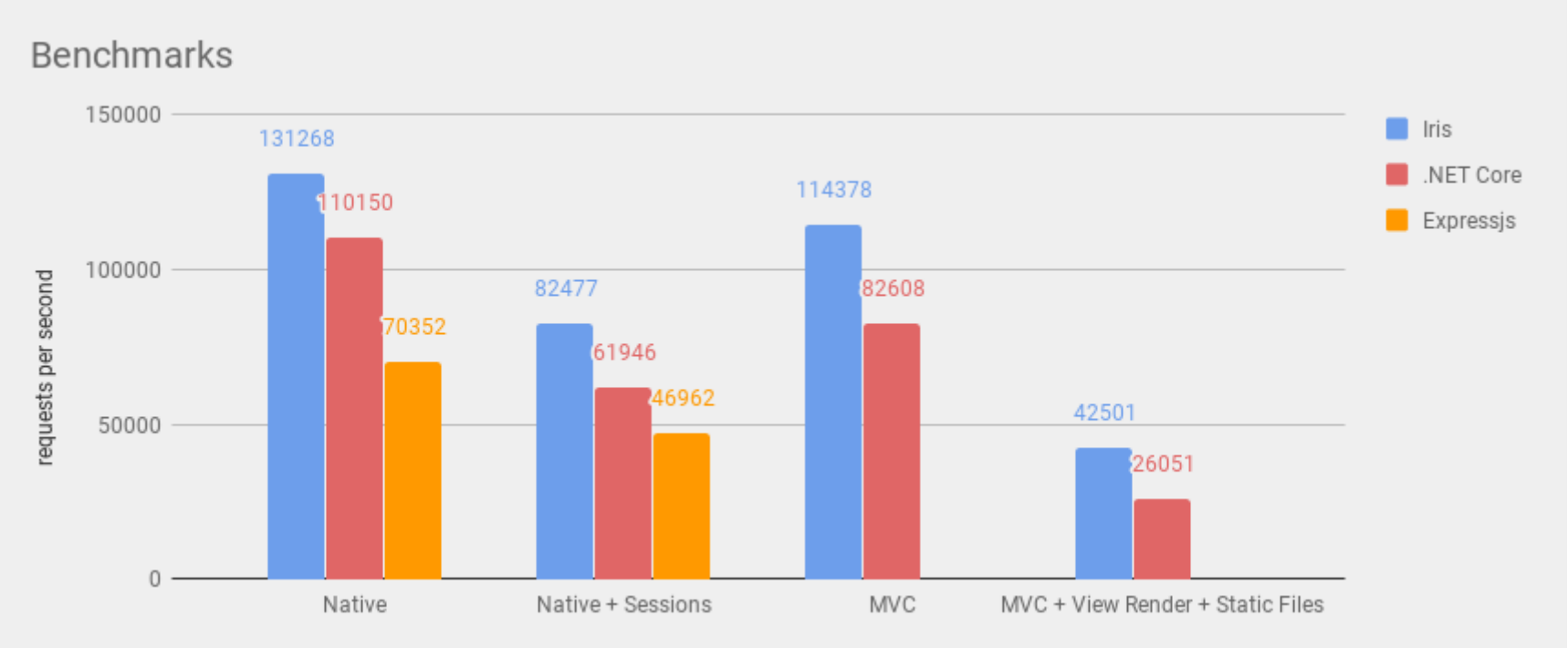
However, I strongly believe that we should not respond back with the same coin, instead, I think we should prove them that compassion and love live inside each one of us, by forgiving them and their actions and look forward for a better place to express ourselves without fear and racism.

Nowadays, Iris is more popular than ever. In fact, their posts, unintentionally, made Iris even more popular than it already was and people started to be very supportive to my dream and this is the reason we are still here.

Do you want to find out how supportive Iris is as a return? You can continue reading our [Support](#) section.

Benchmarks

Iris vs .NET Core vs Expressjs



Updated at: *Monday, 22 October 2018*

Third-party

go (1.11)	muxie (1.0)	6.99 ms	5.51 ms	10.93 ms	23.25 ms	353.24 ms	9725.67
go (1.11)	iris (10.7)	7.08 ms	5.80 ms	11.45 ms	22.85 ms	155.16 ms	4396.00
csharp (7.3)	aspnetcore (2.1)	6.71 ms	5.91 ms	9.62 ms	18.81 ms	270.60 ms	5627.33
go (1.11)	gin (1.3)	8.05 ms	6.27 ms	12.93 ms	27.18 ms	421.50 ms	10821.00
go (1.11)	echo (3.3)	7.82 ms	6.30 ms	12.93 ms	26.62 ms	233.73 ms	6586.00
go (1.11)	beego (1.10)	7.87 ms	6.50 ms	12.96 ms	26.06 ms	163.47 ms	5160.33
go (1.11)	gorilla-mux (1.6)	7.95 ms	6.64 ms	13.20 ms	24.86 ms	203.42 ms	5441.67

Last updated at: 01 March of 2019. Click to the image to view all results. You can run this in your own hardware by following the [steps here](#).

Support

Have you ever be in a position of waiting weeks or months for someone to respond to a github issue of yours? If you choose Iris for your main backend development you will never be treated like a ghost again.

Iris is one of the few github public repositories that offers real support to individuals and collectivities, including companies. Unbeatable free support for the last three years and that is just the beginning of our journey.

In these difficult and restless days we stand beside you. We do not judge bad english writing. We are here for you, no matter who you are.

Navigate through the Iris github [repository](#) to see by yourself the features and the hard work that we putted to improve how the web is built.

Read the latest and greatest features and get migration help

- [HISTORY](#) file is your best friend, it contains the changelog, information about the latest features and changes to the framework.

Did you happen to find a bug?

- Post it at github issues. [Report a bug](#).

Communication

- Do you have any questions or need to speak with someone experienced enough to solve a problem at real-time? Join us to the [Community Chat](#).

Iris is a cross-platform software.

The only requirement is the [Go Programming Language](#), version 1.13 and above.

```
$ cd $YOUR_PROJECT_PATH
$ export G0111MODULE=on
```

Install

```
$ go get github.com/kataras/iris/v12@latest
```

Or edit your project's `go.mod` file.

```
module your_project_name

go 1.13

require (
    github.com/kataras/iris/v12 v12.0.1
)
```

```
$ go build
```

How to update

Here is the go-get command to get the latest and greatest Iris version. Master branch is usually stable enough.

```
$ go get -u github.com/kataras/iris/v12@latest
```

Troubleshooting

If you get a network error during installation please make sure you set a valid [GOPROXY environment variable](#) e.g. `GOPROXY=https://goproxy.io` or `GOPROXY=https://goproxy.cn` .

Continue by reading our [\[\[Getting Started\]\]](#) tutorial.

Getting Started

Iris has an expressive syntax for routing which feels like home. The routing algorithm is powered by the [muxie project](#) which handles requests and matches routes faster than its alternatives like httprouter and gin or echo.

Let's get started without any hassle.

Create an empty file, let's assume its name is `example.go`, then open it and copy-paste the below code.

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.Default()
    app.Use(myMiddleware)

    app.Handle("GET", "/ping", func(ctx iris.Context) {
        ctx.JSON(iris.Map{"message": "pong"})
    })

    // Listens and serves incoming http requests
    // on http://localhost:8080.
    app.Run(iris.Addr(":8080"))
}

func myMiddleware(ctx iris.Context) {
    ctx.Application().Logger().Infof("Runs before %s", ctx.Path())
    ctx.Next()
}
```

Start a terminal session and execute the following.

```
# run example.go and visit http://localhost:8080/ping on browser
$ go run example.go
```

Show me more!

Let's take a small overview of how easy is to get up and running.

```

package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()
    // Load all templates from the "./views" folder
    // where extension is ".html" and parse them
    // using the standard `html/template` package.
    app.RegisterView(iris.HTML("./views", ".html"))

    // Method:    GET
    // Resource:  http://localhost:8080
    app.Get("/", func(ctx iris.Context) {
        // Bind: {{.message}} with "Hello world!"
        ctx.ViewData("message", "Hello world!")
        // Render template file: ./views/hello.html
        ctx.View("hello.html")
    })

    // Method:    GET
    // Resource:  http://localhost:8080/user/42
    //
    // Need to use a custom regexp instead?
    // Easy;
    // Just mark the parameter's type to 'string'
    // which accepts anything and make use of
    // its `regexp` macro function, i.e:
    // app.Get("/user/{id:string regexp(^[0-9]+$)}")
    app.Get("/user/{id:uint64}", func(ctx iris.Context) {
        userID, _ := ctx.Params().GetUint64("id")
        ctx.Writef("User ID: %d", userID)
    })


    // Start the server using a network address.
    app.Run(iris.Addr(":8080"))
}

```

```

<!-- file: ./views/hello.html -->
<html>
<head>
    <title>Hello Page</title>
</head>
<body>
    <h1>{{ .message }}</h1>
</body>
</html>

```



```

$ go run main.go
Now listening on: http://localhost:8080
Application started. Press CTRL+C to shut down.

```

Wanna re-start your app automatically when source code changes happens? Install the [rizla](#) tool and execute `rizla main.go` instead of `go run main.go` .

At the next section we will learn more about [Routing](#).

Host

Listen and Serve

You can start the server(s) listening to any type of `net.Listener` or even `http.Server` instance. The method for initialization of the server should be passed at the end, via `Run` function.

The most common method that Go developers are use to serve their servers are by passing a network address with form of "hostname:ip". With Iris we use the `iris.Addr` which is an `iris.Runner` type

```
// Listening on tcp with network address 0.0.0.0:8080
app.Run(iris.Addr(":8080"))
```

Sometimes you have created a standard net/http server somewhere else in your app and want to use that to serve the Iris web app

```
// Same as before but using a custom http.Server which may being used somewhere else too
app.Run(iris.Server(&http.Server{Addr:":8080"}))
```

The most advanced usage is to create a custom or a standard `net.Listener` and pass that to `app.Run`

```
// Using a custom net.Listener
l, err := net.Listen("tcp4", ":8080")
if err != nil {
    panic(err)
}
app.Run(iris.Listener(l))
```

A more complete example, using the unix-only socket files feature

```
package main

import (
    "os"
    "net"

    "github.com/kataras/iris/v12"
)

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

    // UNIX socket
    if errOs := os.Remove(socketFile); errOs != nil && !os.IsNotExist(errOs) {
        app.Logger().Fatal(errOs)
    }

    l, err := net.Listen("unix", socketFile)

    if err != nil {
        app.Logger().Fatal(err)
    }

    if err = os.Chmod(socketFile, mode); err != nil {
        app.Logger().Fatal(err)
    }

    app.Run(iris.Listener(l))
}
```

UNIX and BSD hosts can take advantage of the reuse port feature

```

package main

import (
    // Package tcplisten provides customizable TCP net.Listener with various
    // performance-related options:
    //
    // - SO_REUSEPORT. This option allows linear scaling server performance
    //   on multi-CPU servers.
    //   See https://www.nginx.com/blog/socket-sharding-nginx-release-1-9-1/ for details.
    //
    // - TCP_DEFER_ACCEPT. This option expects the server reads from the accepted
    //   connection before writing to them.
    //
    // - TCP_FASTOPEN. See https://lwn.net/Articles/508865/ for details.
    "github.com/valyala/tcplisten"

    "github.com/kataras/iris/v12"
)

// go get github.com/valyala/tcplisten
// go run main.go

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

    app.Get("/", func(ctx iris.Context) {
        ctx.HTML("<h1>Hello World!</h1>")
    })

    listenerCfg := tcplisten.Config{
        ReusePort:    true,
        DeferAccept:  true,
        FastOpen:     true,
    }

    l, err := listenerCfg.NewListener("tcp", ":8080")
    if err != nil {
        app.Logger().Fatal(err)
    }

    app.Run(iris.Listener(l))
}

```

HTTP/2 and Secure

If you have signed file keys you can use the `iris.TLS` to serve `https` based on those certification keys

```

// TLS using files
app.Run(iris.TLS("127.0.0.1:443", "mycert.cert", "mykey.key"))

```

The method you should use when your app is ready for **production** is the `iris.AutoTLS` which starts a secure server with automated certifications provided by <https://letsencrypt.org> for **free**

```

// Automatic TLS
app.Run(iris.AutoTLS(":443", "example.com", "admin@example.com"))

```

Any `iris.Runner`

There may be times that you want something very special to listen on, which is not a type of `net.Listener`. You are able to do that by `iris.Raw`, but you're responsible of that method

```

// Using any func() error,
// the responsibility of starting up a listener is up to you with this way,
// for the sake of simplicity we will use the
// ListenAndServe function of the `net/http` package.
app.Run(iris.Raw(&http.Server{Addr:":8080"}).ListenAndServe)

```

Host configurators

All the above forms of listening are accepting a last, variadic argument of `func(*iris.Supervisor)` . This is used to add configurators for that specific host you passed via those functions.

For example let's say that we want to add a callback which is fired when the server is shutdown

```
app.Run(iris.Addr(":8080"), func(h *iris.Supervisor) {
    h.RegisterOnShutdown(func() {
        println("server terminated")
    })
}))
```

You can even do that before `app.Run` method, but the difference is that these host configurators will be executed to all hosts that you may use to serve your web app (via `app.NewHost` we'll see that in a minute)

```
app := iris.New()
app.ConfigureHost(func(h *iris.Supervisor) {
    h.RegisterOnShutdown(func() {
        println("server terminated")
    })
})
app.Run(iris.Addr(":8080"))
```

Access to all hosts that serve your application can be provided by the `Application#Hosts` field, after the `Run` method.

But the most common scenario is that you may need access to the host before the `app.Run` method, there are two ways of gain access to the host supervisor, read below.

We have already saw how to configure all application's hosts by second argument of `app.Run` or `app.ConfigureHost` . There is one more way which suits better for simple scenarios and that is to use the `app.NewHost` to create a new host and use one of its `Serve` or `Listen` functions to start the application via the `iris#Raw` Runner.

Note that this way needs an extra import of the `net/http` package.

Example Code:

```
h := app.NewHost(&http.Server{Addr: ":8080"})
h.RegisterOnShutdown(func(){
    println("server terminated")
})

app.Run(iris.Raw(h.ListenAndServe))
```

Multi hosts

You can serve your Iris web app using more than one server, the `iris.Router` is compatible with the `net/http/Handler` function therefore, as you can understand, it can be used to be adapted at any `net/http` server, however there is an easier way, by using the `app.NewHost` which is also copying all the host configurators and it closes all the hosts attached to the particular web app on `app.Shutdown` .

```
app := iris.New()
app.Get("/", indexHandler)

// run in different goroutine in order to not block the main "goroutine".
go app.Run(iris.Addr(":8080"))
// start a second server which is listening on tcp 0.0.0.0:9090,
// without "go" keyword because we want to block at the last server-run.
app.NewHost(&http.Server{Addr: ":9090"}).ListenAndServe()
```

Shutdown (Gracefully)

Let's continue by learning how to catch CONTROL+C/COMMAND+C or unix kill command and shutdown the server gracefully.

Gracefully Shutdown on CONTROL+C/COMMAND+C or when kill command sent is ENABLED BY-DEFAULT.

In order to manually manage what to do when app is interrupted, we have to disable the default behavior with the option `WithoutInterruptHandler` and register a new interrupt handler (globally, across all possible hosts).

Example code:

```
package main

import (
    "context"
    "time"

    "github.com/kataras/iris/v12"
)

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

    iris.RegisterOnInterrupt(func() {
        timeout := 5 * time.Second
        ctx, cancel := context.WithTimeout(context.Background(), timeout)
        defer cancel()
        // close all hosts
        app.Shutdown(ctx)
    })

    app.Get("/", func(ctx iris.Context) {
        ctx.HTML(" <h1>hi, I just exist in order to see if the server is closed</h1>")
    })

    app.Run(iris.Addr(":8080"), iris.WithoutInterruptHandler)
}
```

Continue reading the [Configuration](#) section to learn about `app.Run` 's second variadic argument.

Automatic Public Domain with TLS

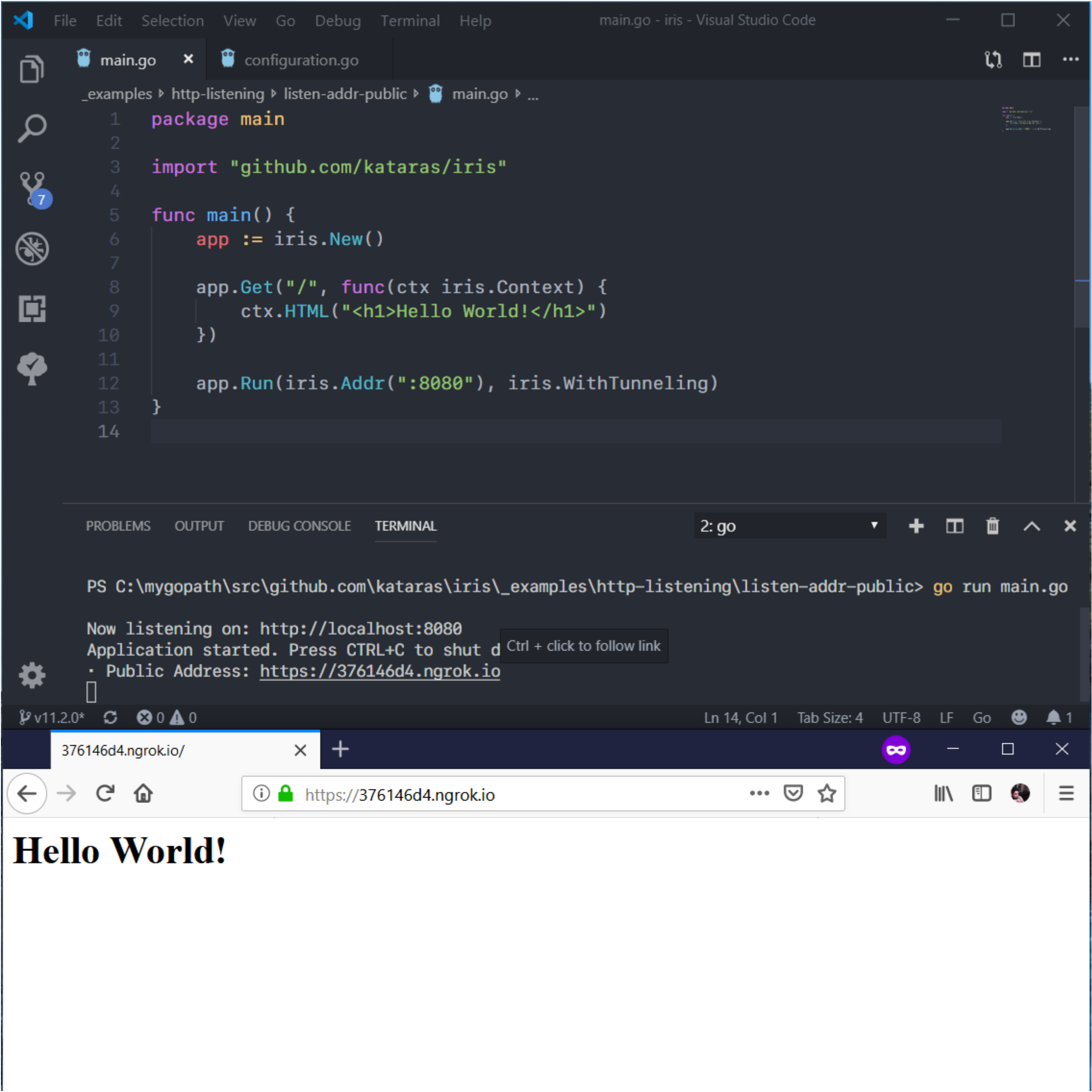
Wouldn't be great to test your web application server in a more "real-world environment" like a public, remote, address instead of localhost?

There are plenty of third-party tools offering such a feature, but in my opinion, the [ngrok](#) one is the best among them. It's popular and tested for years, like Iris.

Iris offers ngrok integration. This feature is simple yet very powerful. It really helps when you want to quickly show your development progress to your colleagues or the project leader at a remote conference.

Follow the steps below to, temporarily, convert your local Iris web server to a public one.

1. Go head and [download ngrok](#), add it to your \$PATH environment variable,
2. Simply pass the `WithTunneling` configurator in your `app.Run` ,
3. You are ready to [GO!](#)



- `ctx.Application().ConfigurationReadOnly().GetVHost()` returns the public domain value. Rarely useful but it's there for you. Most of the times you use relative url paths instead of absolute(or you should to).
- It doesn't matter if ngrok is already running or not, Iris framework is smart enough to use ngrok's [web API](#) to create a tunnel.

Full `Tunneling` configuration:

```
app.Run(iris.Addr(":8080"), iris.WithConfiguration(  
    iris.Configuration{  
        Tunneling: iris.TunnelingConfiguration{  
            AuthToken: "my-ngrok-auth-client-token",  
            Bin:        "/bin/path/for/ngrok",  
            Region:     "eu",  
            WebInterface: "127.0.0.1:4040",  
            Tunnels: []iris.Tunnel{  
                {  
                    Name: "MyApp",  
                    Addr: ":8080",  
                },  
            },  
        },  
    },  
))
```

Read more about [Configuration](#).

Configuration

At the [previous](#) section we've learnt about the first input argument of the `app.Run` , here we will take a look of what the second one is.

Let's start from basics. The `iris.New` function returns an `iris.Application` . This Application value can be configured through its `Configure(...iris.Configurator)` and `Run` methods.

The second optional, variadic argument of `app.Run` method accepts one or more `iris.Configurator` . An `iris.Configurator` is just a type of: `func(app *iris.Application)` . Custom `iris.Configurator` can be passed to modify yours `*iris.Application` as well.

There are built-in `iris.Configurator` s for each of the core [Configuration](#)'s fields, such as `iris.WithoutStartupLog` , `iris.WithCharset("UTF-8")` , `iris.WithOptimizations` and `iris.WithConfiguration(iris.Configuration{...})` functions.

Each “module” like the iris view engine, websockets, sessions and each middleware has its own configurations and options, most of them, separately from the core configuration.

Using the Configuration

The only one configuration structure is the `iris.Configuration` . Let's start by that one which can be passed on the `iris.WithConfiguration` function to make it an `iris.Configurator` .

All of the `iris.Configuration` fields are defaulted to the most common use cases. Iris doesn't need any configuration before its `app.Run` but if you want to make use of a custom `iris.Configurator` before the server runs then you can use the `app.Configure` method to pass the configurator(s) there.

```
config := iris.WithConfiguration(iris.Configuration {
    DisableStartupLog: true,
    Optimizations: true,
    Charset: "UTF-8",
})

app.Run(iris.Addr(":8080"), config)
```

Load from YAML

Using the `iris.YAML("path")` .

File: `iris.yml`

```
FireMethodNotAllowed: true
DisableBodyConsumptionOnUnmarshal: true
TimeFormat: Mon, 01 Jan 2006 15:04:05 GMT
Charset: UTF-8
```

File: `main.go`

```
config := iris.WithConfiguration(iris.YAML("./iris.yml"))
app.Run(iris.Addr(":8080"), config)
```

Load from TOML

Using the `iris.TOML("path")` .

File: `iris.toml`

```
FireMethodNotAllowed = true
DisableBodyConsumptionOnUnmarshal = false
TimeFormat = "Mon, 01 Jan 2006 15:04:05 GMT"
Charset = "UTF-8"

[Other]
  ServerName = "my fancy iris server"
  ServerOwner = "admin@example.com"
```

File: `main.go`

```
config := iris.WithConfiguration(iris.TOML("./iris.toml"))
app.Run(iris.Addr(":8080"), config)
```

Using the functional way

As we already mention, you can pass any number of `iris.Configurator` in the `app.Run` 's second argument. Iris provides an option for each of its `iris.Configuration` 's fields.

```
app.Run(iris.Addr(":8080"), iris.WithoutInterruptHandler,
    iris.WithoutServerError(iris.ErrServerClosed),
    iris.WithoutBodyConsumptionOnUnmarshal,
    iris.WithoutAutoFireStatusCode,
    iris.WithOptimizations,
    iris.WithTimeFormat("Mon, 01 Jan 2006 15:04:05 GMT"),
)
```

Good when you want to change some of the configuration's field. Prefix: "With" or "Without", code editors will help you navigate through all configuration options without even a glitch to the documentation.

Custom values

The `iris.Configuration` contains a field called `Other map[string]interface{}` which accepts any custom `key:value` option, therefore you can use that field to pass specific values that your app expects based on the custom requirements.

```
app.Run(iris.Addr(":8080"),
    iris.WithOtherValue("ServerName", "my amazing iris server"),
    iris.WithOtherValue("ServerOwner", "admin@example.com"),
)
```

You can access those fields via `app.ConfigurationReadOnly` .

```
serverName := app.ConfigurationReadOnly().Other["MyServerName"]
serverOwner := app.ConfigurationReadOnly().Other["ServerOwner"]
```

Access Configuration from Context

Inside a handler, retrieve those fields using the following:

```
ctx.Application().ConfigurationReadOnly()
```


Routing

The Handler type

A Handler, as the name implies, handle requests.

A Handler responds to an HTTP request. It writes reply headers and data to the `Context.ResponseWriter()` and then return. Returning signals that the request is finished; it is not valid to use the Context after or concurrently with the completion of the Handler call.

Depending on the HTTP client software, HTTP protocol version, and any intermediaries between the client and the iris server, it may not be possible to read from the `Context.Request().Body` after writing to the `context.ResponseWriter()`. Cautious handlers should read the `Context.Request().Body` first, and then reply.

Except for reading the body, handlers should not modify the provided Context.

If Handler panics, the server (the caller of Handler) assumes that the effect of the panic was isolated to the active request. It recovers the panic, logs a stack trace to the server error log and hangs up the connection.

```
type Handler func(iris.Context)
```

Once the handler is registered, we can use the returned `Route` instance to give a name to the handler registration for easier debugging or match relative paths in views. For more information, checkout the [Reverse lookups](#) section.

Behavior

Iris' default behavior is to accept and register routes with paths like `/api/user`, without a trailing slash. If a client tries to reach `$your_host/api/user/` then the Iris router will automatically permant redirect this to `$your_host/api/user` in order to be handled by the registered route. This is the modern way to design APIs.

However, if you want to **disable path correction** for the requested resources you can pass the `iris.WithoutPathCorrection` option of the iris [Configuration](#) to your `app.Run`. Example:

```
// [app := iris.New...]
// [...]

app.Run(iris.Addr(":8080"), iris.WithoutPathCorrection)
```

If you want to keep the same handler and route for `/api/user` and `/api/user/` paths **without redirection**(common scenario) use just the `iris.WithoutPathCorrectionRedirection` option instead:

```
app.Run(iris.Addr(":8080"), iris.WithoutPathCorrectionRedirection)
```

API

All HTTP methods are supported, developers can also register handlers on the same path with different methods.

The first parameter is the HTTP Method, second parameter is the request path of the route, third variadic parameter should contain one or more `iris.Handler` executed by the registered order when a client requests for that specific resouce path from the server.

Example code:

```
app := iris.New()

app.Handle("GET", "/contact", func(ctx iris.Context) {
    ctx.HTML("<h1> Hello from /contact </h1>")
})
```

In order to make things easier for the end-developer, iris provides method helpers for all HTTP Methods. The first parameter is the request path of the route, second variadic parameter should contains one or more `iris.Handler` executed by the registered order when a user requests for that specific resouce path from the server.

Example code:

```

app := iris.New()

// Method: "GET"
app.Get("/", handler)

// Method: "POST"
app.Post("/", handler)

// Method: "PUT"
app.Put("/", handler)

// Method: "DELETE"
app.Delete("/", handler)

// Method: "OPTIONS"
app.Options("/", handler)

// Method: "TRACE"
app.Trace("/", handler)

// Method: "CONNECT"
app.Connect("/", handler)

// Method: "HEAD"
app.Head("/", handler)

// Method: "PATCH"
app.Patch("/", handler)

// register the route for all HTTP Methods
app.Any("/", handler)

func handler(ctx iris.Context){
    ctx.Writef("Hello from method: %s and path: %s\n", ctx.Method(), ctx.Path())
}

```

Offline Routes

There is one special method in Iris that you can use too. It's called `None` and you can use it to hide a route from outsiders but still able to call it from other route's handlers through the `Context.Exec` method. Each API Handle method returns the Route value back. A Route the `IsOnline` method which reports back the current state of that route. You can change the **state** of the route from **offline** to **online** and visa-versa through its `Route.Method` field's value. Of course each change of the router at serve-time requires an `app.RefreshRouter()` call which is safe to use. Take a look below a more complete example:

```

// file: main.go
package main

import (
    "github.com/kataras/iris/v12"
)

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

    none := app.None("/invisible/{username}", func(ctx iris.Context) {
        ctx.Writef("Hello %s with method: %s", ctx.Params().Get("username"), ctx.Method())

        if from := ctx.Values().GetString("from"); from != "" {
            ctx.Writef("\nI see that you're coming from %s", from)
        }
    })

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

        if none.IsOnline() {
            none.Method = iris.MethodNone
        } else {
            none.Method = iris.MethodGet
        }

        // refresh re-builds the router at serve-time in order to be notified for its new routes.
        app.RefreshRouter()
    })

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

        if none.IsOnline() {
            none.Method = iris.MethodNone
        } else {
            none.Method = iris.MethodGet
        }

        // refresh re-builds the router at serve-time in order to
        // be notified for its new routes.
        app.RefreshRouter()
    })

    app.Get("/execute", func(ctx iris.Context) {
        if !none.IsOnline() {
            ctx.Values().Set("from", "/execute with offline access")
            ctx.Exec("NONE", "/invisible/iris")
            return
        }

        // same as navigating to "http://localhost:8080/invisible/iris"
        // when /change has being invoked and route state changed
        // from "offline" to "online"
        ctx.Values().Set("from", "/execute")
        // values and session can be
        // shared when calling Exec from a "foreign" context.
        //     ctx.Exec("NONE", "/invisible/iris")
        // or after "/change":
        ctx.Exec("GET", "/invisible/iris")
    })

    app.Run(iris.Addr(":8080"))
}

```

1. `go run main.go`
2. Open a browser at `http://localhost:8080/invisible/iris` and you'll see that you get a `404 not found` error,
3. however the `http://localhost:8080/execute` will be able to execute that route.
4. Now, if you navigate to the `http://localhost:8080/change` and refresh the `/invisible/iris` tab you'll see that you can see it.

Grouping Routes

A set of routes that are being grouped by path prefix can (optionally) share the same middleware handlers and template layout. A group can have a nested group too.

`.Party` is being used to group routes, developers can declare an unlimited number of (nested) groups.

Example code:

```
app := iris.New()

users := app.Party("/users", myAuthMiddlewareHandler)

// http://localhost:8080/users/42/profile
users.Get("/{id:uint64}/profile", userProfileHandler)
// http://localhost:8080/users/messages/1
users.Get("/messages/{id:uint64}", userMessageHandler)
```

The same could be also written using the `PartyFunc` method which accepts the child router(the Party).

```
app := iris.New()

app.PartyFunc("/users", func(users iris.Party) {
    users.Use(myAuthMiddlewareHandler)

    // http://localhost:8080/users/42/profile
    users.Get("/{id:uint64}/profile", userProfileHandler)
    // http://localhost:8080/users/messages/1
    users.Get("/messages/{id:uint64}", userMessageHandler)
})
```

Path Parameters

Unlike other routers you'd seen, the Iris' one can handle various route paths without confliction between them.

Matches only GET `"/`.

```
app.Get("/", indexHandler)
```

Matches all GET requests prefixed with `"/assets/**/*"`, it's a wildcard with `ctx.Params().Get("asset")` equals to any following path after the `/assets/`.

```
app.Get("/assets/{asset:path}", assetsWildcardHandler)
```

Matches all GET requests prefixed with `"/profile/"` and followed by a single path part.

```
app.Get("/profile/{username:string}", userHandler)
```

Matches only GET `"/profile/me"` and it does not conflict with `/profile/{username:string}` or any root wildcard `/{root:path}`.

```
app.Get("/profile/me", userHandler)
```

Matches all GET requests prefixed with `/users/` and followed by a number which should be equal or higher than 1.

```
app.Get("/user/{userid:int min(1)}", getUserHandler)
```

Matches all DELETE requests prefixed with `/users/` and following by a number which should be equal or higher than 1.

```
app.Delete("/user/{userid:int min(1)}", deleteUserHandler)
```

Matches all GET requests except the ones that are already handled by other routes. For example in this case by the above routes; `/`, `/assets/{asset:path}`, `/profile/{username}`, `"/profile/me"`, `/user/{userid:int ...}`. It does not conflict with the rest of the routes(!).

```
app.Get("/{root:path}", rootWildcardHandler)
```

Matches all GET requests of:

1. /u/abcd maps to :alphabetical (if :alphabetical registered otherwise :string)
2. /u/42 maps to :uint (if :uint registered otherwise :int)
3. /u/-1 maps to :int (if :int registered otherwise :string)
4. /u/abcd123 maps to :string

```
app.Get("/u/{username:string}", func(ctx iris.Context) {
    ctx.Writef("username (string): %s", ctx.Params().Get("username"))
})

app.Get("/u/{id:int}", func(ctx iris.Context) {
    ctx.Writef("id (int): %d", ctx.Params().GetIntDefault("id", 0))
})

app.Get("/u/{uid:uint}", func(ctx iris.Context) {
    ctx.Writef("uid (uint): %d", ctx.Params().GetUintDefault("uid", 0))
})

app.Get("/u/{firstname:alphabetical}", func(ctx iris.Context) {
    ctx.Writef("firstname (alphabetical): %s", ctx.Params().Get("firstname"))
})
```

Matches all GET requests of `/abctenchars.xml` and `/abcdtenchars` respectfully.

```
app.Get("/{alias:string regexp(^[a-z0-9]{1,10}\\.xml$})", PanoXML)
app.Get("/{alias:string regexp(^[a-z0-9]{1,10}$})", Tour)
```

You may wonder what the `{id:uint64}` or `:path` or `min(1)` are. They are (typed) dynamic path parameters and functions can be registered on them. Learn more by reading the [Path Parameter Types](#).

Path Parameter Types

Iris has the easiest and the most powerful routing process you have ever met.

Iris has its own interpeter for route's path syntax, parsing and evaluation (yes, like a programming language!).

It's fast, how? It calculates its needs and if not any special regexp needed then it just registers the route with the low-level path syntax, otherwise it pre-compiles the regexp and adds the necessary middleware(s). That means that you have zero performance cost compared to other routers or web frameworks.

Parameters

A path parameter's name should contain only alphabetical letters. Numbers or symbols like '_' are NOT allowed.

Do not confuse `ctx.Params()` with `ctx.Values()` .

- Path parameter's values can be retrieved from `ctx.Params()` .
- Context's local storage that can be used to communicate between handlers and middleware(s) can be stored to `ctx.Values()` .

The built-in available parameter types can be found at the following table.

Param Type	Go Type	Validation	Retrieve Helper
<code>:string</code>	string	anything (single path segment)	<code>Params().Get</code>
<code>:int</code>	int	-9223372036854775808 to 9223372036854775807 (x64) or -2147483648 to 2147483647 (x32), depends on the host arch	<code>Params().GetInt</code>
<code>:int8</code>	int8	-128 to 127	<code>Params().GetInt8</code>
<code>:int16</code>	int16	-32768 to 32767	<code>Params().GetInt16</code>
<code>:int32</code>	int32	-2147483648 to 2147483647	<code>Params().GetInt32</code>
<code>:int64</code>	int64	-9223372036854775808 to 9223372036854775807	<code>Params().GetInt64</code>
<code>:uint</code>	uint	0 to 18446744073709551615 (x64) or 0 to 4294967295 (x32), depends on the host arch	<code>Params().GetUint</code>
<code>:uint8</code>	uint8	0 to 255	<code>Params().GetUint8</code>
<code>:uint16</code>	uint16	0 to 65535	<code>Params().GetUint16</code>
<code>:uint32</code>	uint32	0 to 4294967295	<code>Params().GetUint32</code>
<code>:uint64</code>	uint64	0 to 18446744073709551615	<code>Params().GetUint64</code>
<code>:bool</code>	bool	"1" or "t" or "T" or "TRUE" or "true" or "True" or "0" or "f" or "F" or "FALSE" or "false" or "False"	<code>Params().GetBool</code>
<code>:alphabetical</code>	string	lowercase or uppercase letters	<code>Params().Get</code>
<code>:file</code>	string	lowercase or uppercase letters, numbers, underscore (<code>_</code>), dash (<code>-</code>), point (<code>.</code>) and no spaces or other special characters that are not valid for filenames	<code>Params().Get</code>
<code>:path</code>	string	anything, can be separated by slashes (path segments) but should be the last part of the route path	<code>Params().Get</code>

Usage:

```
app.Get("/users/{id:uint64}", func(ctx iris.Context){
    id := ctx.Params().GetUint64Default("id", 0)
    // [...]
})
```

Built-in Func	Param Types
<code>regexp</code> (expr string)	<code>:string</code>
<code>prefix</code> (prefix string)	<code>:string</code>
<code>suffix</code> (suffix string)	<code>:string</code>
<code>contains</code> (s string)	<code>:string</code>
<code>min</code> (minValue int or int8 or int16 or int32 or int64 or uint8 or uint16 or uint32 or uint64 or float32 or float64)	<code>:string(char length)</code> , <code>:int</code> , <code>:int8</code> , <code>:int16</code> , <code>:int32</code> , <code>:int64</code> , <code>:uint</code> , <code>:uint8</code> , <code>:uint16</code> , <code>:uint32</code> , <code>:uint64</code>
<code>max</code> (maxValue int or int8 or int16 or int32 or int64 or uint8 or uint16 or uint32 or uint64 or float32 or float64)	<code>:string(char length)</code> , <code>:int</code> , <code>:int8</code> , <code>:int16</code> , <code>:int32</code> , <code>:int64</code> , <code>:uint</code> , <code>:uint8</code> , <code>:uint16</code> , <code>:uint32</code> , <code>:uint64</code>
<code>range</code> (minValue, maxValue int or int8 or int16 or int32 or int64 or uint8 or uint16 or uint32 or uint64 or float32 or float64)	<code>:int</code> , <code>:int8</code> , <code>:int16</code> , <code>:int32</code> , <code>:int64</code> , <code>:uint</code> , <code>:uint8</code> , <code>:uint16</code> , <code>:uint32</code> , <code>:uint64</code>

Usage:

```
app.Get("/profile/{name:alphabetical max(255)}", func(ctx iris.Context){
    name := ctx.Params().Get("name")
    // len(name) <=255 otherwise this route will fire 404 Not Found
    // and this handler will not be executed at all.
})
```

Do It Yourself:

The `RegisterFunc` can accept any function that returns a `func(paramValue string) bool` . Or just a `func(string) bool` . If the validation fails then it will fire `404` or whatever status code the `else` keyword has.


```
latLonExpr := "^-?[0-9]{1,3}(?:\\. [0-9]{1,10})? $"
latLonRegex, _ := regexp.Compile(latLonExpr)

// Register your custom argument-less macro function to the :string param type.
// MatchString is a type of func(string) bool, so we use it as it is.
app.Macros().Get("string").RegisterFunc("coordinate", latLonRegex.MatchString)

app.Get("/coordinates/{lat:string coordinate()}/{lon:string coordinate()}",
func(ctx iris.Context) {
    ctx.Writef("Lat: %s | Lon: %s", ctx.Params().Get("lat"), ctx.Params().Get("lon"))
})
```

Register your custom macro function which accepts two int arguments.

```
app.Macros().Get("string").RegisterFunc("range",
func(minLength, maxLength int) func(string) bool {
    return func(paramValue string) bool {
        return len(paramValue) >= minLength && len(paramValue) <= maxLength
    }
})

app.Get("/limitchar/{name:string range(1,200) else 400}", func(ctx iris.Context) {
    name := ctx.Params().Get("name")
    ctx.Writef(`Hello %s | the name should be between 1 and 200 characters length
otherwise this handler will not be executed`, name)
})
```

Register your custom macro function which accepts a slice of strings `[...]` .

```
app.Macros().Get("string").RegisterFunc("has",
func(validNames []string) func(string) bool {
    return func(paramValue string) bool {
        for _, validName := range validNames {
            if validName == paramValue {
                return true
            }
        }

        return false
    }
})

app.Get("/static_validation/{name:string has([kataras,maropoulos])}",
func(ctx iris.Context) {
    name := ctx.Params().Get("name")
    ctx.Writef(`Hello %s | the name should be "kataras" or "maropoulos"
otherwise this handler will not be executed`, name)
})
```

Example Code:

```

func main() {
    app := iris.Default()

    // This handler will match /user/john but will not match neither /user/ or /user.
    app.Get("/user/{name}", func(ctx iris.Context) {
        name := ctx.Params().Get("name")
        ctx.Writef("Hello %s", name)
    })

    // This handler will match /users/42
    // but will not match /users/-1 because uint should be bigger than zero
    // neither /users or /users/.
    app.Get("/users/{id:uint64}", func(ctx iris.Context) {
        id := ctx.Params().GetUint64Default("id", 0)
        ctx.Writef("User with ID: %d", id)
    })

    // However, this one will match /user/john/send and also /user/john/everything/else/here
    // but will not match /user/john neither /user/john/.
    app.Post("/user/{name:string}/{action:path}", func(ctx iris.Context) {
        name := ctx.Params().Get("name")
        action := ctx.Params().Get("action")
        message := name + " is " + action
        ctx.WriteString(message)
    })

    app.Run(iris.Addr(":8080"))
}

```

When parameter type is missing then it defaults to the `string` one, therefore `{name:string}` and `{name}` refers to the same exactly thing.

Reverse Lookups

As mentioned in the [Routing](#) chapter, Iris provides several handler registration methods, each of which returns a `Route` instance.

Route naming

Route naming is easy, since we just call the returned `*Route` with a `Name` field to define a name:

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()
    // define a function
    h := func(ctx iris.Context) {
        ctx.HTML("<b>Hi</b1>")
    }

    // handler registration and naming
    home := app.Get("/", h)
    home.Name = "home"
    // or
    app.Get("/about", h).Name = "about"
    app.Get("/page/{id}", h).Name = "page"

    app.Run(iris.Addr(":8080"))
}
```

Route reversing AKA generating URLs from the route name

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

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

Above code would generate the following output:

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

Using route names in code

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

- `GetRoutes` function to get all registered routes
- `GetRoute(routeName string)` method to retrieve a route by name
- `URL(routeName string, paramValues ...interface{})` method to generate url string based on supplied parameters
- `Path(routeName string, paramValues ...interface{})` method to generate just the path (without host and protocol) portion of the URL based on provided values

Middleware

When we talk about Middleware in Iris we're talking about running code before and/or after our main handler code in a HTTP request lifecycle. For example, logging middleware might write the incoming request details to a log, then call the handler code, before writing details about the response to the log. One of the cool things about middleware is that these units are extremely flexible and reusable.

A middleware is just a **Handler** form of `func(ctx iris.Context)`, the middleware is being executed when the previous middleware calls the `ctx.Next()`, this can be used for authentication, i.e: if request authenticated then call `ctx.Next()` to process with the rest of the chain of handlers in the request otherwise fire an error response.

Writing a middleware

```
package main

import "github.com/kataras/iris/v12"

func main() {
    app := iris.New()
    // or app.Use(before) and app.Done(after).
    app.Get("/", before, mainHandler, after)
    app.Run(iris.Addr(":8080"))
}

func before(ctx iris.Context) {
    shareInformation := "this is a sharable information between handlers"

    requestPath := ctx.Path()
    println("Before the mainHandler: " + requestPath)

    ctx.Values().Set("info", shareInformation)
    ctx.Next() // execute the next handler, in this case the main one.
}

func after(ctx iris.Context) {
    println("After the mainHandler")
}

func mainHandler(ctx iris.Context) {
    println("Inside mainHandler")

    // take the info from the "before" handler.
    info := ctx.Values().GetString("info")

    // write something to the client as a response.
    ctx.HTML("<h1>Response</h1>")
    ctx.HTML("<br/> Info: " + info)

    ctx.Next() // execute the "after".
}
```

```
$ go run main.go # and navigate to the http://localhost:8080
Now listening on: http://localhost:8080
Application started. Press CTRL+C to shut down.
Before the mainHandler: /
Inside mainHandler
After the mainHandler
```

Globally

```

package main

import "github.com/kataras/iris/v12"

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

    // register our routes.
    app.Get("/", indexHandler)
    app.Get("/contact", contactHandler)

    // Order of those calls does not matter,
    // `UseGlobal` and `DoneGlobal` are applied to existing routes
    // and future routes also.
    //
    // Remember: the `Use` and `Done` are applied to the current party's and its children,
    // so if we used the `app.Use/Done` before the routes registration
    // it would work like UseGlobal/DoneGlobal in this case,
    // because the `app` is the root "Party".
    app.UseGlobal(before)
    app.DoneGlobal(after)

    app.Run(iris.Addr(":8080"))
}

func before(ctx iris.Context) {
    // [...]
}

func after(ctx iris.Context) {
    // [...]
}

func indexHandler(ctx iris.Context) {
    // write something to the client as a response.
    ctx.HTML("<h1>Index</h1>")

    ctx.Next() // execute the "after" handler registered via `Done`.
}

func contactHandler(ctx iris.Context) {
    // write something to the client as a response.
    ctx.HTML("<h1>Contact</h1>")

    ctx.Next() // execute the "after" handler registered via `Done`.
}

```

You could also use the `ExecutionRules` to force Done handlers to be executed without the need of `ctx.Next()` in your route handlers, do it like this:

```

app.SetExecutionRules(iris.ExecutionRules{
    // Begin: ...
    // Main: ...
    Done: iris.ExecutionOptions{Force: true},
})

```

Convert `http.Handler/HandlerFunc`

However you are not limited to them - you are free to use any third-party middleware that is compatible with the [net/http](#) package.

Iris, unlike others, is 100% compatible with the standards and that's why the majority of big companies that adapt Go to their workflow, like a very famous US Television Network, trust Iris; it's up-to-date and it will be always aligned with the std `net/http` package which is modernized by the Go Authors on each new release of the Go Programming Language.

Any third-party middleware that written for `net/http` is compatible with Iris using the `iris.FromStd(aThirdPartyMiddleware)`. Remember, `ctx.ResponseWriter()` and `ctx.Request()` returns the same `net/http` input arguments of an [http.Handler](#).

- From `func(w http.ResponseWriter, r *http.Request, next http.HandlerFunc)`
- From `http.Handler` or `http.HandlerFunc`

- [From func\(http.HandlerFunc\) http.HandlerFunc](#)

Here is a list of some handlers made specifically for Iris:

Built-in

Middleware	Example
basic authentication	iris/_examples/authentication/basicauth
Google reCAPTCHA	iris/_examples/miscellaneous/recaptcha
localization and internationalization	iris/_examples/miscellaneous/i81n
request logger	iris/_examples/http_request/request-logger
profiling (pprof)	iris/_examples/miscellaneous/pprof
recovery	iris/_examples/miscellaneous/recover

Community made

Most of the experimental handlers are ported to work with *iris*'s handler form, from third-party sources.

Middleware	Description	Example
jwt	Middleware checks for a JWT on the <code>Authorization</code> header on incoming requests and decodes it.	iris-contrib/middleware/jwt/_example
cors	HTTP Access Control.	iris-contrib/middleware/cors/_example
secure	Middleware that implements a few quick security wins.	iris-contrib/middleware/secure/_example
tollbooth	Generic middleware to rate-limit HTTP requests.	iris-contrib/middleware/tollbooth/_examples/limit-handler
cloudwatch	AWS cloudwatch metrics middleware.	iris-contrib/middleware/cloudwatch/_example
new relic	Official New Relic Go Agent .	iris-contrib/middleware/newrelic/_example
prometheus	Easily create metrics endpoint for the prometheus instrumentation tool	iris-contrib/middleware/prometheus/_example
casbin	An authorization library that supports access control models like ACL, RBAC, ABAC	iris-contrib/middleware/casbin/_examples
raven	Sentry client in Go	iris-contrib/middleware/raven/_example
csrf	Cross-Site Request Forgery Protection	iris-contrib/middleware/csrf/_example

Handle HTTP errors

You can define your own handlers when a specific http error code occurs.

Error codes are the http status codes that are bigger than or equal to 400, like 404 not found and 500 internal server.

Example code:

```
package main

import "github.com/kataras/iris/v12"

func main(){
    app := iris.New()
    app.OnErrorCode(iris.StatusNotFound, notFound)
    app.OnErrorCode(iris.StatusInternalServerError, internalServerError)
    // to register a handler for all "error"
    // status codes(kataras/iris/context.StatusCodeNotSuccessful)
    // defaults to < 200 || >= 400:
    // app.OnAnyErrorCode(handler)
    app.Get("/", index)
    app.Run(iris.Addr(":8080"))
}

func notFound(ctx iris.Context) {
    // when 404 then render the template
    // $views_dir/errors/404.html
    ctx.View("errors/404.html")
}

func internalServerError(ctx iris.Context) {
    ctx.WriteString("Oups something went wrong, try again")
}

func index(ctx iris.Context) {
    ctx.View("index.html")
}
```

Learn more about [View](#).

The Problem type

Iris has builtin support for the [Problem Details for HTTP APIs](#).

The `Context.Problem` writes a JSON or XML problem response. Behaves exactly like `Context.JSON` but with default `ProblemOptions.JSON` indent of " " and a response content type of "application/problem+json" instead.

Use the `options.RenderXML` and `XML` fields to change this behavior and send a response of content type "application/problem+xml" instead.

```

func newProductProblem(productName, detail string) iris.Problem {
    return iris.NewProblem().
        // The type URI, if relative it automatically convert to absolute.
        Type("/product-error").
        // The title, if empty then it gets it from the status code.
        Title("Product validation problem").
        // Any optional details.
        Detail(detail).
        // The status error code, required.
        Status(iris.StatusBadRequest).
        // Any custom key-value pair.
        Key("productName", productName)
        // Optional cause of the problem, chain of Problems.
        // .Cause(other iris.Problem)
}

```

```

func fireProblem(ctx iris.Context) {
    // Response like JSON but with indent of " " and
    // content type of "application/problem+json"
    ctx.Problem(newProductProblem("product name", "problem details"),
        iris.ProblemOptions{
            // Optional JSON renderer settings.
            JSON: iris.JSON{
                Indent: " ",
            },
            // OR
            // Render as XML:
            // RenderXML: true,
            // XML: iris.XML{Indent: " "},
            // Sets the "Retry-After" response header.
            //
            // Can accept:
            // time.Time for HTTP-Date,
            // time.Duration, int64, float64, int for seconds
            // or string for date or duration.
            // Examples:
            // time.Now().Add(5 * time.Minute),
            // 300 * time.Second,
            // "5m",
            //
            RetryAfter: 300,
            // A function that, if specified, can dynamically set
            // retry-after based on the request.
            // Useful for ProblemOptions reusability.
            // Overrides the RetryAfter field.
            //
            // RetryAfterFunc: func(iris.Context) interface{} { [...] }
        })
}

```

Outputs "application/problem+json"

```

{
  "type": "https://host.domain/product-error",
  "status": 400,
  "title": "Product validation problem",
  "detail": "problem error details",
  "productName": "product name"
}

```

When `RenderXML` is set to `true` then the response will look be rendered as XML instead.

Outputs "application/problem+xml"

```
<Problem>
  <Type>https://host.domain/product-error</Type>
  <Status>400</Status>
  <Title>Product validation problem</Title>
  <Detail>problem error details</Detail>
  <ProductName>product name</ProductName>
</Problem>
```

Full example can be found at [_examples/routing/http-errors](#).

Subdomains

Iris has the simplest known form for subdomains registration to a single application. Of course you can always use nginx or caddy for management in production.

Subdomains are separated into two categories: **static** and **dynamic/wildcard**.

- Static : when you know the subdomain, i.e : `analytics.mydomain.com`
- Wildcard : when you don't know the subdomain but you know that it's before a particular subdomain or root domain, i.e : `user_created.mydomain.com` , `otheruser.mydomain.com` like the `username.github.io`

We use the `Subdomain` and `WildcardSubdomain` methods of an `iris.Party` or `iris.Application` to register subdomains.

The `Subdomain` method returns a new `Party` which is responsible to register routes to this specific "subdomain".

The only difference from a regular `Party` is that if called from a child party then the subdomain will be prepended to the path instead of appended. So if `app.Subdomain("admin").Subdomain("panel")` then the result is: `"panel.admin."` .

```
Subdomain(subdomain string, middleware ...Handler) Party
```

The `WildcardSubdomain` method returns a new `Party` which is responsible to register routes to a dynamic, wildcard(ed) subdomain. A dynamic subdomain is a subdomain which can handle any subdomain requests. Server will accept any subdomain (if not static subdomain found) and it will search and execute the handlers of this `Party` .

```
WildcardSubdomain(middleware ...Handler) Party
```

Example Code:

```
// [app := iris.New...]
admin := app.Subdomain("admin")

// admin.mydomain.com
admin.Get("/", func(ctx iris.Context) {
    ctx.Writef("INDEX FROM admin.mydomain.com")
})

// admin.mydomain.com/hey
admin.Get("/hey", func(ctx iris.Context) {
    ctx.Writef("HEY FROM admin.mydomain.com/hey")
})

// [other routes here...]

app.Run(iris.Addr("mydomain.com:80"))
```

For local development you'll have to edit your hosts, for example in windows operating system open the `C:\Windows\System32\Drivers\etc\hosts` file and append:

```
127.0.0.1 mydomain.com
127.0.0.1 admin.mydomain.com
```

To prove that subdomains works like any other regular `Party` you can also register a subdomain using the alternative method below:

```
adminSubdomain:= app.Party("admin.")
// or
adminAnalayticsSubdomain := app.Party("admin.analytics.")
// or for a dynamic one:
anySubdomain := app.Party("*.")
```

There is also an `iris.Application` method which allows to register a global redirection rule for subdomains as well.

The `SubdomainRedirect` sets (or adds if used more than one time) a router wrapper which redirects(StatusMovedPermanently) a (sub)domain to another subdomain or to the root domain as fast as possible, before the execution of the route's handler(s).

It receives two arguments, they are the from and to/target locations, 'from' can be a wildcard subdomain as well (app.WildcardSubdomain()) 'to' is not allowed to be a wildcard for obvious reasons, 'from' can be the root domain(app) when the 'to' is not the root domain and visa-versa.

```
SubdomainRedirect(from, to Party) Party
```

Usage

```
www := app.Subdomain("www")
app.SubdomainRedirect(app, www)
```

The above will redirect all `http(s)://mydomain.com/%anypath%` to `http(s)://www.mydomain.com/%anypath%`.

The `Context` offers four main methods when working with subdomains that may be helpful for you.

```
// Host returns the host part of the current url.  
Host() string  
// Subdomain returns the subdomain of this request, if any.  
// Note that this is a fast method which does not cover all cases.  
Subdomain() (subdomain string)  
// IsWWW returns true if the current subdomain (if any) is www.  
IsWWW() bool  
// FullRquestURI returns the full URI,  
// including the scheme, the host and the relative requested path/resource.  
FullRequestURI() string
```

Usage

```
func info(ctx iris.Context) {  
    method := ctx.Method()  
    subdomain := ctx.Subdomain()  
    path := ctx.Path()  
  
    ctx.Writef("\nInfo\n\n")  
    ctx.Writef("Method: %s\nSubdomain: %s\nPath: %s", method, subdomain, path)  
}
```

Wrap the Router

You may never need this one but it's here in case.

There are times that you may need to override or decide whether the Router will be executed on an incoming request. If you've any previous experience with the `net/http` and other web frameworks this function will be familiar with you (it has the form of a `net/http` middleware, but instead of accepting the next handler it accepts the Router as a function to be executed or not).

```
// WrapperFunc is used as an expected input parameter signature
// for the WrapRouter. It's a "low-level" signature which is compatible
// with the net/http.
// It's being used to run or no run the router based on a custom logic.
type WrapperFunc func(w http.ResponseWriter, r *http.Request, router http.HandlerFunc)

// WrapRouter adds a wrapper on the top of the main router.
// Usually it's useful for third-party middleware
// when need to wrap the entire application with a middleware like CORS.
//
// Developers can add more than one wrappers,
// those wrappers' execution comes from last to first.
// That means that the second wrapper will wrap the first, and so on.
//
// Before build.
func WrapRouter(wrapperFunc WrapperFunc)
```

The Router searches for its routes based on the `Subdomain` , `HTTP Method` and its dynamic `Path` . A Router Wrapper can override that behavior and execute custom code.

In this example you'll just see one use case of `.WrapRouter`. You can use the `.WrapRouter` to add custom logic when or when not the router should be executed in order to execute the registered routes' handlers. This is just for the proof of concept, you can skip this tutorial.

Example Code:

```

package main

import (
    "net/http"
    "strings"

    "github.com/kataras/iris/v12"
)

func newApp() *iris.Application {
    app := iris.New()

    app.OnErrorCode(iris.StatusNotFound, func(ctx iris.Context) {
        ctx.HTML("<b>Resource Not found</b>")
    })

    app.Get("/profile/{username}", func(ctx iris.Context) {
        ctx.Writef("Hello %s", ctx.Params().Get("username"))
    })

    app.HandleDir("/", "./public")

    myOtherHandler := func(ctx iris.Context) {
        ctx.Writef("inside a handler which is fired manually by our custom router wrapper")
    }

    // wrap the router with a native net/http handler.
    // if url does not contain any "." (i.e: .css, .js...)
    // (depends on the app , you may need to add more file-server exceptions),
    // then the handler will execute the router that is responsible for the
    // registered routes (look "/" and "/profile/{username}")
    // if not then it will serve the files based on the root "/" path.
    app.WrapRouter(func(w http.ResponseWriter, r *http.Request, router http.HandlerFunc) {
        path := r.URL.Path

        if strings.HasPrefix(path, "/other") {
            // acquire and release a context in order to use it to execute
            // our custom handler
            // remember: we use net/http.Handler because here
            // we are in the "low-level", before the router itself.
            ctx := app.ContextPool.Acquire(w, r)
            myOtherHandler(ctx)
            app.ContextPool.Release(ctx)
            return
        }

        // else continue serving routes as usual.
        router.ServeHTTP(w, r)
    })

    return app
}

func main() {
    app := newApp()

    // http://localhost:8080
    // http://localhost:8080/index.html
    // http://localhost:8080/app.js
    // http://localhost:8080/css/main.css
    // http://localhost:8080/profile/anyusername
    // http://localhost:8080/other/random
    app.Run(iris.Addr(":8080"))

    // Note: In this example we just saw one use case,

```

```
// you may want to .WrapRouter or .Downgrade in order to
// bypass the Iris' default router, i.e:
// you can use that method to setup custom proxies too.
}
```

There is not much to say here, it's just a function wrapper which accepts the native response writer and request and the next handler which is the Iris' Router itself, it's being or not executed whether is called or not, **it's a middleware for the whole Router**.

Override Context

In this section you will learn how to override the existing [Context](#)'s methods.

The [Context](#) is an interface. However as you probably know, when using other frameworks you don't have that functionality of overriding even if it's used as an interface. With Iris you can **attach** your implementation to the **context pool** itself using the `app.ContextPool.Attach` method.

1. Let's get started by importing the `"github.com/kataras/iris/v12/context"` which is required here.
2. Secondly, create your own Context implementation.
3. Add the `Do` and `Next` methods where they just call the `context.Do` and `context.Next` package-level functions.
4. Use the Application's `ContextPool` to set it as the Context implementation that should be used for the route's handlers.

Example Code:

Please read the *comments* too.

```

package main

import (
    "reflect"

    "github.com/kataras/iris/v12"
    // 1.
    "github.com/kataras/iris/v12/context"
)

// 2.
// Create your own custom Context, put any fields you'll need.
type MyContext struct {
    // Embed the `iris.Context` -
    // It's totally optional but you will need this if you
    // don't want to override all the context's methods!
    iris.Context
}

// Optionally: validate MyContext implements iris.Context on compile-time.
var _ iris.Context = &MyContext{}

// 3.
func (ctx *MyContext) Do(handlers context.Handlers) {
    context.Do(ctx, handlers)
}

// 3.
func (ctx *MyContext) Next() {
    context.Next(ctx)
}

// [Override any context's method you want here...]
// Like the HTML below:

func (ctx *MyContext) HTML(htmlContents string) (int, error) {
    ctx.Application().Logger().Infof("Executing .HTML function from MyContext")

    ctx.ContentType("text/html")
    return ctx.WriteString(htmlContents)
}

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

    // 4.
    app.ContextPool.Attach(func() iris.Context {
        return &MyContext{
            // If you use the embedded Context,
            // call the `context.NewContext` to create one:
            Context: context.NewContext(app),
        }
    })

    // Register a view engine on .html files inside the ./view/** directory.
    app.RegisterView(iris.HTML("./view", ".html"))

    // Register your route, as you normally do
    app.Handle("GET", "/", recordWhichContextForExample,
        func(ctx iris.Context) {
            // use the context's overridden HTML method.
            ctx.HTML("<h1> Hello from my custom context's HTML! </h1>")
        })
}

```

```
// This will be executed by the
// MyContext.Context embedded default context
// when MyContext is not directly define the View function by itself.
app.Handle("GET", "/hi/{firstname:alphabetical}",recordWhichContextForExample,
func(ctx iris.Context) {
    firstname := ctx.Values().GetString("firstname")

    ctx.ViewData("firstname", firstname)
    ctx.Gzip(true)

    ctx.View("hi.html")
})

app.Run(iris.Addr(":8080"))
}

// Should always print "($PATH) Handler is executing from 'MyContext'"
func recordWhichContextForExample(ctx iris.Context) {
    ctx.Application().Logger().Infof("(%s) Handler is executing from: '%s'",
        ctx.Path(), reflect.TypeOf(ctx).Elem().Name())

    ctx.Next()
}
```

All Context Methods

Here is a full list of methods that the `iris.Context` provides.


```

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

    // Unmarshaler is the interface implemented by types that can unmarshal any raw data.
    // TIP INFO: Any pointer to a value which implements the BodyDecoder can be override the unmarshaler.
    Unmarshaler interface {
        Unmarshal(data []byte, outPtr interface{}) error
    }

    // UnmarshalerFunc a shortcut for the Unmarshaler interface
    //
    // See 'Unmarshaler' and 'BodyDecoder' for more.
    UnmarshalerFunc func(data []byte, outPtr interface{}) error
)

// Unmarshal parses the X-encoded data and stores the result in the value pointed to by v.
// Unmarshal uses the inverse of the encodings that Marshal uses, allocating maps,
// slices, and pointers as necessary.
func (u UnmarshalerFunc) Unmarshal(data []byte, v interface{}) error {
    return u(data, v)
}

// Context is the middle-man server's "object" for the clients.
//
// A New context is being acquired from a sync.Pool on each connection.
// The Context is the most important thing on the iris's http flow.
//
// Developers send responses to the client's request through a Context.
// Developers get request information from the client's request a Context.
type Context interface {
    // BeginRequest is executing once for each request
    // it should prepare the (new or acquired from pool) context's fields for the new request.
    //
    // To follow the iris' flow, developer should:
    // 1. reset handlers to nil
    // 2. reset values to empty
    // 3. reset sessions to nil
    // 4. reset response writer to the http.ResponseWriter
    // 5. reset request to the *http.Request
    // and any other optional steps, depends on dev's application type.
    BeginRequest(http.ResponseWriter, *http.Request)
    // EndRequest is executing once after a response to the request was sent and this context is useless or released.
    //
    // To follow the iris' flow, developer should:
    // 1. flush the response writer's result
    // 2. release the response writer
    // and any other optional steps, depends on dev's application type.
    EndRequest()
}

```

```

// ResponseWriter returns an http.ResponseWriter compatible response writer, as expected.
ResponseWriter() ResponseWriter
// ResetResponseWriter should change or upgrade the Context's ResponseWriter.
ResetResponseWriter(ResponseWriter)

// Request returns the original *http.Request, as expected.
Request() *http.Request
// ResetRequest sets the Context's Request,
// It is useful to store the new request created by a std *http.Request#WithContext() into Iris' Context.
// Use `ResetRequest` when for some reason you want to make a full
// override of the *http.Request.
// Note that: when you just want to change one of each fields you can use the Request() which returns a pointer to Request
// so the changes will have affect without a full override.
// Usage: you use a native http handler which uses the standard "context" package
// to get values instead of the Iris' Context#Values():
// r := ctx.Request()
// stdCtx := context.WithValue(r.Context(), key, val)
// ctx.ResetRequest(r.WithContext(stdCtx)).
ResetRequest(r *http.Request)

// SetCurrentRouteName sets the route's name internally,
// in order to be able to find the correct current "read-only" Route when
// end-developer calls the `GetCurrentRoute()` function.
// It's being initialized by the Router, if you change that name
// manually nothing really happens except that you'll get other
// route via `GetCurrentRoute()`.
// Instead, to execute a different path
// from this context you should use the `Exec` function
// or change the handlers via `SetHandlers/AddHandler` functions.
SetCurrentRouteName(currentRouteName string)
// GetCurrentRoute returns the current registered "read-only" route that
// was being registered to this request's path.
GetCurrentRoute() RouteReadOnly

// Do calls the SetHandlers(handlers)
// and executes the first handler,
// handlers should not be empty.
//
// It's used by the router, developers may use that
// to replace and execute handlers immediately.
Do(Handlers)

// AddHandler can add handler(s)
// to the current request in serve-time,
// these handlers are not persisted to the router.
//
// Router is calling this function to add the route's handler.
// If AddHandler called then the handlers will be inserted
// to the end of the already-defined route's handler.
//
AddHandler(...Handler)
// SetHandlers replaces all handlers with the new.
SetHandlers(Handlers)
// Handlers keeps tracking of the current handlers.
Handlers() Handlers

// HandlerIndex sets the current index of the
// current context's handlers chain.
// If -1 passed then it just returns the
// current handler index without change the current index.
//
// Look Handlers(), Next() and StopExecution() too.
HandlerIndex(n int) (currentIndex int)
// Proceed is an alternative way to check if a particular handler
// has been executed and called the `ctx.Next` function inside it.

```

```

// This is useful only when you run a handler inside
// another handler. It just checks for before index and the after index.
//
// A usecase example is when you want to execute a middleware
// inside controller's `BeginRequest` that calls the `ctx.Next` inside it.
// The Controller looks the whole flow (BeginRequest, method handler, EndRequest)
// as one handler, so `ctx.Next` will not be reflected to the method handler
// if called from the `BeginRequest`.
//
// Although `BeginRequest` should NOT be used to call other handlers,
// the `BeginRequest` has been introduced to be able to set
// common data to all method handlers before their execution.
// Controllers can accept middleware(s) from the MVC's Application's Router as normally.
//
// That said let's see an example of `ctx.Proceed`:
//
// var authMiddleware = basicauth.New(basicauth.Config{
//     Users: map[string]string{
//         "admin": "password",
//     },
// })
//
// func (c *UsersController) BeginRequest(ctx iris.Context) {
//     if !ctx.Proceed(authMiddleware) {
//         ctx.StopExecution()
//     }
// }
//
// This Get() will be executed in the same handler as `BeginRequest`,
// internally controller checks for `ctx.StopExecution`.
// So it will not be fired if BeginRequest called the `StopExecution`.
// func(c *UsersController) Get() []models.User {
//     return c.Service.GetAll()
// }
//
// Alternative way is `!ctx.IsStopped()` if middleware make use of the `ctx.StopExecution()` on failure.
Proceed(Handler) bool
// HandlerName returns the current handler's name, helpful for debugging.
HandlerName() string
// HandlerFileLine returns the current running handler's function source file and line information.
// Useful mostly when debugging.
HandlerFileLine() (file string, line int)
// RouteName returns the route name that this handler is running on.
// Note that it will return empty on not found handlers.
RouteName() string
// Next calls all the next handler from the handlers chain,
// it should be used inside a middleware.
//
// Note: Custom context should override this method in order to be able to pass its own iris.Context implementation.
Next()
// NextOr checks if chain has a next handler, if so then it executes it
// otherwise it sets a new chain assigned to this Context based on the given handler(s)
// and executes its first handler.
//
// Returns true if next handler exists and executed, otherwise false.
//
// Note that if no next handler found and handlers are missing then
// it sends a Status Not Found (404) to the client and it stops the execution.
NextOr(handlers ...Handler) bool
// NextOrNotFound checks if chain has a next handler, if so then it executes it
// otherwise it sends a Status Not Found (404) to the client and stops the execution.
//
// Returns true if next handler exists and executed, otherwise false.
NextOrNotFound() bool
// NextHandler returns (it doesn't execute) the next handler from the handlers chain.
//
// Use .Skip() to skip this handler if needed to execute the next of this returning handler.
NextHandler() Handler

```

```

// Skip skips/ignores the next handler from the handlers chain,
// it should be used inside a middleware.
Skip()
// StopExecution if called then the following .Next calls are ignored,
// as a result the next handlers in the chain will not be fire.
StopExecution()
// IsStopped checks and returns true if the current position of the Context is 255,
// means that the StopExecution() was called.
IsStopped() bool
// OnConnectionClose registers the "cb" function which will fire (on its own goroutine, no need to be registered goroutine)
// when the underlying connection has gone away.
//
// This mechanism can be used to cancel long operations on the server
// if the client has disconnected before the response is ready.
//
// It depends on the `http#CloseNotify`.
// CloseNotify may wait to notify until Request.Body has been
// fully read.
//
// After the main Handler has returned, there is no guarantee
// that the channel receives a value.
//
// Finally, it reports whether the protocol supports pipelines (HTTP/1.1 with pipelines disabled is not supported).
// The "cb" will not fire for sure if the output value is false.
//
// Note that you can register only one callback for the entire request handler chain/per route.
//
// Look the `ResponseWriter#CloseNotifier` for more.
OnConnectionClose(fnGoroutine func()) bool
// OnClose registers the callback function "cb" to the underline connection closing event using the `Context#OnConnectionClose`
// and also in the end of the request handler using the `ResponseWriter#SetBeforeFlush`.
// Note that you can register only one callback for the entire request handler chain/per route.
//
// Look the `Context#OnConnectionClose` and `ResponseWriter#SetBeforeFlush` for more.
OnClose(cb func())

// +-----+
// | Current "user/request" storage |
// | and share information between the handlers - Values(). |
// | Save and get named path parameters - Params() |
// +-----+

// Params returns the current url's named parameters key-value storage.
// Named path parameters are being saved here.
// This storage, as the whole Context, is per-request lifetime.
Params() *RequestParams

// Values returns the current "user" storage.
// Named path parameters and any optional data can be saved here.
// This storage, as the whole Context, is per-request lifetime.
//
// You can use this function to Set and Get local values
// that can be used to share information between handlers and middleware.
Values() *memstore.Store
// Translate is the i18n (localization) middleware's function,
// it calls the Values().Get(ctx.Application().Configuration.ReadOnly().GetTranslateFunctionContextKey())
// to execute the translate function and return the localized text value.
//
// Example: https://github.com/kataras/iris/tree/master/_examples/miscellaneous/i18n
Translate(format string, args ...interface{}) string

// +-----+
// | Path, Host, Subdomain, IP, Headers etc... |
// +-----+

// Method returns the request.Method, the client's http method to the server.

```

```

Method() string
// Path returns the full request path,
// escaped if EnablePathEscape config field is true.
Path() string
// RequestPath returns the full request path,
// based on the 'escape'.
RequestPath(escape bool) string
// Host returns the host part of the current url.
Host() string
// Subdomain returns the subdomain of this request, if any.
// Note that this is a fast method which does not cover all cases.
Subdomain() (subdomain string)
// IsWWW returns true if the current subdomain (if any) is www.
IsWWW() bool
// FullRquestURI returns the full URI,
// including the scheme, the host and the relative requested path/resource.
FullRequestURI() string
// RemoteAddr tries to parse and return the real client's request IP.
//
// Based on allowed headers names that can be modified from Configuration.RemoteAddrHeaders.
//
// If parse based on these headers fail then it will return the Request's `RemoteAddr` field
// which is filled by the server before the HTTP handler.
//
// Look `Configuration.RemoteAddrHeaders`,
//      `Configuration.WithRemoteAddrHeader(...)` ,
//      `Configuration.WithoutRemoteAddrHeader(...)` for more.
RemoteAddr() string
// GetHeader returns the request header's value based on its name.
GetHeader(name string) string
// IsAjax returns true if this request is an 'ajax request'( XMLHttpRequest)
//
// There is no a 100% way of knowing that a request was made via Ajax.
// You should never trust data coming from the client, they can be easily overcome by spoofing.
//
// Note that "X-Requested-With" Header can be modified by any client(because of "X-"),
// so don't rely on IsAjax for really serious stuff,
// try to find another way of detecting the type(i.e, content type),
// there are many blogs that describe these problems and provide different kind of solutions,
// it's always depending on the application you're building,
// this is the reason why this `IsAjax` is simple enough for general purpose use.
//
// Read more at: https://developer.mozilla.org/en-US/docs/AJAX
// and https://xhr.spec.whatwg.org/
IsAjax() bool
// IsMobile checks if client is using a mobile device(phone or tablet) to communicate with this server.
// If the return value is true that means that the http client using a mobile
// device to communicate with the server, otherwise false.
//
// Keep note that this checks the "User-Agent" request header.
IsMobile() bool
// GetReferrer extracts and returns the information from the "Referer" header as specified
// in https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy
// or by the URL query parameter "referrer".
GetReferrer() Referrer
// +-----+
// | Headers helpers                               |
// +-----+

// Header adds a header to the response writer.
Header(name string, value string)

// ContentType sets the response writer's header key "Content-Type" to the 'cType'.
ContentType(cType string)
// GetContentType returns the response writer's header value of "Content-Type"
// which may, set before with the 'ContentType'.

```



```

GetContentType() string
// GetContentType returns the request's header value of "Content-Type".
GetContentTypeRequested() string

// GetContentLength returns the request's header value of "Content-Length".
// Returns 0 if header was unable to be found or its value was not a valid number.
GetContentLength() int64

// StatusCode sets the status code header to the response.
// Look .`GetStatusCode` too.
StatusCode(statusCode int)
// GetStatusCode returns the current status code of the response.
// Look `StatusCode` too.
GetStatusCode() int

// Redirect sends a redirect response to the client
// to a specific url or relative path.
// accepts 2 parameters string and an optional int
// first parameter is the url to redirect
// second parameter is the http status should send,
// default is 302 (StatusFound),
// you can set it to 301 (Permant redirect)
// or 303 (StatusSeeOther) if POST method,
// or StatusTemporaryRedirect(307) if that's nessecery.
Redirect(urlToRedirect string, statusHeader ...int)

// +-----+
// | Various Request and Post Data          |
// +-----+

// URLParam returns true if the url parameter exists, otherwise false.
URLParamExists(name string) bool
// URLParamDefault returns the get parameter from a request,
// if not found then "def" is returned.
URLParamDefault(name string, def string) string
// URLParam returns the get parameter from a request, if any.
URLParam(name string) string
// URLParamTrim returns the url query parameter with trailing white spaces removed from a request.
URLParamTrim(name string) string
// URLParamTrim returns the escaped url query parameter from a request.
URLParamEscape(name string) string
// URLParamInt returns the url query parameter as int value from a request,
// returns -1 and an error if parse failed.
URLParamInt(name string) (int, error)
// URLParamIntDefault returns the url query parameter as int value from a request,
// if not found or parse failed then "def" is returned.
URLParamIntDefault(name string, def int) int
// URLParamInt32Default returns the url query parameter as int32 value from a request,
// if not found or parse failed then "def" is returned.
URLParamInt32Default(name string, def int32) int32
// URLParamInt64 returns the url query parameter as int64 value from a request,
// returns -1 and an error if parse failed.
URLParamInt64(name string) (int64, error)
// URLParamInt64Default returns the url query parameter as int64 value from a request,
// if not found or parse failed then "def" is returned.
URLParamInt64Default(name string, def int64) int64
// URLParamFloat64 returns the url query parameter as float64 value from a request,
// returns -1 and an error if parse failed.
URLParamFloat64(name string) (float64, error)
// URLParamFloat64Default returns the url query parameter as float64 value from a request,
// if not found or parse failed then "def" is returned.
URLParamFloat64Default(name string, def float64) float64
// URLParamBool returns the url query parameter as boolean value from a request,
// returns an error if parse failed or not found.
URLParamBool(name string) (bool, error)
// URLParams returns a map of GET query parameters separated by comma if more than one

```

```

// it returns an empty map if nothing found.
URLParams() map[string]string

// FormValueDefault returns a single parsed form value by its "name",
// including both the URL field's query parameters and the POST or PUT form data.
//
// Returns the "def" if not found.
FormValueDefault(name string, def string) string
// FormValue returns a single parsed form value by its "name",
// including both the URL field's query parameters and the POST or PUT form data.
FormValue(name string) string
// FormValues returns the parsed form data, including both the URL
// field's query parameters and the POST or PUT form data.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris#WithPostMaxMemory` configurator at main configuration passed on `app.Run`'s second argument.
//
// NOTE: A check for nil is necessary.
FormValues() map[string][]string

// PostValueDefault returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name".
//
// If not found then "def" is returned instead.
PostValueDefault(name string, def string) string
// PostValue returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name"
PostValue(name string) string
// PostValueTrim returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", without trailing spaces.
PostValueTrim(name string) string
// PostValueInt returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as int.
//
// If not found returns -1 and a non-nil error.
PostValueInt(name string) (int, error)
// PostValueIntDefault returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as int.
//
// If not found returns or parse errors the "def".
PostValueIntDefault(name string, def int) int
// PostValueInt64 returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as float64.
//
// If not found returns -1 and a no-nil error.
PostValueInt64(name string) (int64, error)
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as int64.
//
// If not found or parse errors returns the "def".
PostValueInt64Default(name string, def int64) int64
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as float64.
//
// If not found returns -1 and a non-nil error.
PostValueFloat64(name string) (float64, error)
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as float64.
//
// If not found or parse errors returns the "def".
PostValueFloat64Default(name string, def float64) float64
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as bool.
//
// If not found or value is false, then it returns false, otherwise true.
PostValueBool(name string) (bool, error)

```

```

// PostValues returns all the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name" as a string slice.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris#WithPostMaxMemory` configurator at main configuration passed on `app.Run`'s second argument.
PostValues(name string) []string
// FormFile returns the first uploaded file that received from the client.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris#WithPostMaxMemory` configurator at main configuration passed on `app.Run`'s second argument.
//
// Example: https://github.com/kataras/iris/tree/master/_examples/http_request/upload-file
FormFile(key string) (multipart.File, *multipart.FileHeader, error)
// UploadFormFiles uploads any received file(s) from the client
// to the system physical location "destDirectory".
//
// The second optional argument "before" gives caller the chance to
// modify the *multipart.FileHeader before saving to the disk,
// it can be used to change a file's name based on the current request,
// all FileHeader's options can be changed. You can ignore it if
// you don't need to use this capability before saving a file to the disk.
//
// Note that it doesn't check if request body streamed.
//
// Returns the copied length as int64 and
// a not nil error if at least one new file
// can't be created due to the operating system's permissions or
// http.ErrMissingFile if no file received.
//
// If you want to receive & accept files and manage them manually you can use the `context#FormFile`
// instead and create a copy function that suits your needs, the below is for generic usage.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris#WithPostMaxMemory` configurator at main configuration passed on `app.Run`'s second argument.
//
// See `FormFile` to a more controlled to receive a file.
//
// Example: https://github.com/kataras/iris/tree/master/_examples/http_request/upload-files
UploadFormFiles(destDirectory string, before ...func(Context, *multipart.FileHeader)) (n int64, err error)

// +-----+
// | Custom HTTP Errors                                |
// +-----+

// NotFound emits an error 404 to the client, using the specific custom error error handler.
// Note that you may need to call ctx.StopExecution() if you don't want the next handlers
// to be executed. Next handlers are being executed on iris because you can alt the
// error code and change it to a more specific one, i.e
// users := app.Party("/users")
// users.Done(func(ctx iris.Context){ if ctx.StatusCode() == 400 { /* custom error code for /users */ })
NotFound()

// +-----+
// | Body Readers                                    |
// +-----+

// SetMaxRequestBodySize sets a limit to the request body size
// should be called before reading the request body from the client.
SetMaxRequestBodySize(limitOverBytes int64)

// GetBody reads and returns the request body.
// The default behavior for the http request reader is to consume the data readen
// but you can change that behavior by passing the `WithoutBodyConsumptionOnUnmarshal` iris option.
//
// However, whenever you can use the `ctx.Request().Body` instead.

```



```

GetBody() ([]byte, error)
// UnmarshalBody reads the request's body and binds it to a value or pointer of any type.
// Examples of usage: context.ReadJSON, context.ReadXML.
//
// Example: https://github.com/kataras/iris/blob/master/_examples/http_request/read-custom-via-unmarshaler/main.go
//
// UnmarshalBody does not check about gzipped data.
// Do not rely on compressed data incoming to your server. The main reason is: https://en.wikipedia.org/wiki/Zip_bomb
// However you are still free to read the `ctx.Request().Body io.Reader` manually.
UnmarshalBody(outPtr interface{}, unmarshaler Unmarshaler) error
// ReadJSON reads JSON from request's body and binds it to a pointer of a value of any json-valid type.
//
// Example: https://github.com/kataras/iris/blob/master/_examples/http_request/read-json/main.go
ReadJSON(jsonObjectPtr interface{}) error
// ReadXML reads XML from request's body and binds it to a pointer of a value of any xml-valid type.
//
// Example: https://github.com/kataras/iris/blob/master/_examples/http_request/read-xml/main.go
ReadXML(xmlObjectPtr interface{}) error
// ReadForm binds the formObject with the form data
// it supports any kind of type, including custom structs.
// It will return nothing if request data are empty.
// The struct field tag is "form".
//
// Example: https://github.com/kataras/iris/blob/master/_examples/http_request/read-form/main.go
ReadForm(formObject interface{}) error
// ReadQuery binds the "ptr" with the url query string. The struct field tag is "url".
//
// Example: https://github.com/kataras/iris/blob/master/_examples/http_request/read-query/main.go
ReadQuery(ptr interface{}) error
//
// +-----+
// | Body (raw) Writers                               |
// +-----+

// Write writes the data to the connection as part of an HTTP reply.
//
// If WriteHeader has not yet been called, Write calls
// WriteHeader(http.StatusOK) before writing the data. If the Header
// does not contain a Content-Type line, Write adds a Content-Type set
// to the result of passing the initial 512 bytes of written data to
// DetectContentType.
//
// Depending on the HTTP protocol version and the client, calling
// Write or WriteHeader may prevent future reads on the
// Request.Body. For HTTP/1.x requests, handlers should read any
// needed request body data before writing the response. Once the
// headers have been flushed (due to either an explicit Flusher.Flush
// call or writing enough data to trigger a flush), the request body
// may be unavailable. For HTTP/2 requests, the Go HTTP server permits
// handlers to continue to read the request body while concurrently
// writing the response. However, such behavior may not be supported
// by all HTTP/2 clients. Handlers should read before writing if
// possible to maximize compatibility.
Write(body []byte) (int, error)
// Writef formats according to a format specifier and writes to the response.
//
// Returns the number of bytes written and any write error encountered.
Writef(format string, args ...interface{}) (int, error)
// WriteString writes a simple string to the response.
//
// Returns the number of bytes written and any write error encountered.
WriteString(body string) (int, error)

// SetLastModified sets the "Last-Modified" based on the "modtime" input.
// If "modtime" is zero then it does nothing.
//
// It's mostly internally on core/router and context packages.

```

```

//
// Note that modtime.UTC() is being used instead of just modtime, so
// you don't have to know the internals in order to make that works.
SetLastModified(modtime time.Time)
// CheckIfModifiedSince checks if the response is modified since the "modtime".
// Note that it has nothing to do with server-side caching.
// It does those checks by checking if the "If-Modified-Since" request header
// sent by client or a previous server response header
// (e.g with WriteWithExpiration or HandleDir or Favicon etc.)
// is a valid one and it's before the "modtime".
//
// A check for !modtime && err == nil is necessary to make sure that
// it's not modified since, because it may return false but without even
// had the chance to check the client-side (request) header due to some errors,
// like the HTTP Method is not "GET" or "HEAD" or if the "modtime" is zero
// or if parsing time from the header failed.
//
// It's mostly used internally, e.g. `context#WriteWithExpiration`.
//
// Note that modtime.UTC() is being used instead of just modtime, so
// you don't have to know the internals in order to make that works.
CheckIfModifiedSince(modtime time.Time) (bool, error)
// WriteNotModified sends a 304 "Not Modified" status code to the client,
// it makes sure that the content type, the content length headers
// and any "ETag" are removed before the response sent.
//
// It's mostly used internally on core/router/fs.go and context methods.
WriteNotModified()
// WriteWithExpiration works like `Write` but it will check if a resource is modified,
// based on the "modtime" input argument,
// otherwise sends a 304 status code in order to let the client-side render the cached content.
WriteWithExpiration(body []byte, modtime time.Time) (int, error)
// StreamWriter registers the given stream writer for populating
// response body.
//
// Access to context's and/or its' members is forbidden from writer.
//
// This function may be used in the following cases:
//
//     * if response body is too big (more than iris.LimitRequestBodySize(if set)).
//     * if response body is streamed from slow external sources.
//     * if response body must be streamed to the client in chunks.
//     (aka `http server push`).
//
// receives a function which receives the response writer
// and returns false when it should stop writing, otherwise true in order to continue
StreamWriter(writer func(w io.Writer) bool)

// +-----+
// | Body Writers with compression          |
// +-----+
// ClientSupportsGzip retruns true if the client supports gzip compression.
ClientSupportsGzip() bool
// WriteGzip accepts bytes, which are compressed to gzip format and sent to the client.
// returns the number of bytes written and an error ( if the client doesn' supports gzip compression)
// You may re-use this function in the same handler
// to write more data many times without any troubles.
WriteGzip(b []byte) (int, error)
// TryWriteGzip accepts bytes, which are compressed to gzip format and sent to the client.
// If client does not supprots gzip then the contents are written as they are, uncompressed.
TryWriteGzip(b []byte) (int, error)
// GzipResponseWriter converts the current response writer into a response writer
// which when its .Write called it compress the data to gzip and writes them to the client.
//
// Can be also disabled with its .Disable and .ResetBody to rollback to the usual response writer.
GzipResponseWriter() *GzipResponseWriter

```

```

// Gzip enables or disables (if enabled before) the gzip response writer,if the client
// supports gzip compression, so the following response data will
// be sent as compressed gzip data to the client.
Gzip(enable bool)

// +-----+
// | Rich Body Content Writers/Renderers |
// +-----+

// ViewLayout sets the "layout" option if and when .View
// is being called afterwards, in the same request.
// Useful when need to set or/and change a layout based on the previous handlers in the chain.
//
// Note that the 'layoutTplFile' argument can be set to iris.NoLayout || view.NoLayout
// to disable the layout for a specific view render action,
// it disables the engine's configuration's layout property.
//
// Look .ViewData and .View too.
//
// Example: https://github.com/kataras/iris/tree/master/_examples/view/context-view-data/
ViewLayout(layoutTplFile string)
// ViewData saves one or more key-value pair in order to be passed if and when .View
// is being called afterwards, in the same request.
// Useful when need to set or/and change template data from previous hanadlers in the chain.
//
// If .View's "binding" argument is not nil and it's not a type of map
// then these data are being ignored, binding has the priority, so the main route's handler can still decide.
// If binding is a map or context.Map then these data are being added to the view data
// and passed to the template.
//
// After .View, the data are not destroyed, in order to be re-used if needed (again, in the same request as everything else)
// to clear the view data, developers can call:
// ctx.Set(ctx.Application().ConfigurationReadOnly().GetViewDataContextKey(), nil)
//
// If 'key' is empty then the value is added as it's (struct or map) and developer is unable to add other value.
//
// Look .ViewLayout and .View too.
//
// Example: https://github.com/kataras/iris/tree/master/_examples/view/context-view-data/
ViewData(key string, value interface{})
// GetViewData returns the values registered by `context#ViewData`.
// The return value is `map[string]interface{}`, this means that
// if a custom struct registered to ViewData then this function
// will try to parse it to map, if failed then the return value is nil
// A check for nil is always a good practise if different
// kind of values or no data are registered via `ViewData`.
//
// Similarly to `viewData := ctx.Values().Get("iris.viewData")` or
// `viewData := ctx.Values().Get(ctx.Application().ConfigurationReadOnly().GetViewDataContextKey())`.
GetViewData() map[string]interface{}
// View renders a template based on the registered view engine(s).
// First argument accepts the filename, relative to the view engine's Directory and Extension,
// i.e: if directory is "./templates" and want to render the "./templates/users/index.html"
// then you pass the "users/index.html" as the filename argument.
//
// The second optional argument can receive a single "view model"
// that will be binded to the view template if it's not nil,
// otherwise it will check for previous view data stored by the `ViewData`
// even if stored at any previous handler(middleware) for the same request.
//
// Look .ViewData` and .ViewLayout too.
//
// Examples: https://github.com/kataras/iris/tree/master/_examples/view
View(filename string, optionalViewModel ...interface{}) error

// Binary writes out the raw bytes as binary data.

```

```

Binary(data []byte) (int, error)
// Text writes out a string as plain text.
Text(format string, args ...interface{}) (int, error)
// HTML writes out a string as text/html.
HTML(format string, args ...interface{}) (int, error)
// JSON marshals the given interface object and writes the JSON response.
JSON(v interface{}, options ...JSON) (int, error)
// JSONP marshals the given interface object and writes the JSON response.
JSONP(v interface{}, options ...JSONP) (int, error)
// XML marshals the given interface object and writes the XML response.
XML(v interface{}, options ...XML) (int, error)
// Markdown parses the markdown to html and renders its result to the client.
Markdown(markdownB []byte, options ...Markdown) (int, error)
// YAML parses the "v" using the yaml parser and renders its result to the client.
YAML(v interface{}) (int, error)
// +-----+
// | Serve files                                |
// +-----+

// ServeContent serves content, headers are autose
// receives three parameters, it's low-level function, instead you can use .ServeFile(string,bool)/SendFile(string,string)
//
//
// You can define your own "Content-Type" with `context#ContentType`, before this function call.
//
// This function doesn't support resuming (by range),
// use ctx.SendFile or router's `HandleDir` instead.
ServeContent(content io.ReadSeeker, filename string, modtime time.Time, gzipCompression bool) error
// ServeFile serves a file (to send a file, a zip for example to the client you should use the `SendFile` instead)
// receives two parameters
// filename/path (string)
// gzipCompression (bool)
//
// You can define your own "Content-Type" with `context#ContentType`, before this function call.
//
// This function doesn't support resuming (by range),
// use ctx.SendFile or router's `HandleDir` instead.
//
// Use it when you want to serve dynamic files to the client.
ServeFile(filename string, gzipCompression bool) error
// SendFile sends file for force-download to the client
//
// Use this instead of ServeFile to 'force-download' bigger files to the client.
SendFile(filename string, destinationName string) error

// +-----+
// | Cookies                                |
// +-----+

// SetCookie adds a cookie.
// Use of the "options" is not required, they can be used to amend the "cookie".
//
// Example: https://github.com/kataras/iris/tree/master/_examples/cookies/basic
SetCookie(cookie *http.Cookie, options ...CookieOption)
// SetCookieKV adds a cookie, requires the name(string) and the value(string).
//
// By default it expires at 2 hours and it's added to the root path,
// use the `CookieExpires` and `CookiePath` to modify them.
// Alternatively: ctx.SetCookie(&http.Cookie{...})
//
// If you want to set custom the path:
// ctx.SetCookieKV(name, value, iris.CookiePath("/custom/path/cookie/will/be/stored"))
//
// If you want to be visible only to current request path:
// ctx.SetCookieKV(name, value, iris.CookieCleanPath/iris.CookiePath(""))
// More:

```



```

// iris.CookieExpires(time.Duration)
// iris.CookieHTTPOnly(false)
//
// Example: https://github.com/kataras/iris/tree/master/_examples/cookies/basic
SetCookieKV(name, value string, options ...CookieOption)
// GetCookie returns cookie's value by its name
// returns empty string if nothing was found.
//
// If you want more than the value then:
// cookie, err := ctx.Request().Cookie("name")
//
// Example: https://github.com/kataras/iris/tree/master/_examples/cookies/basic
GetCookie(name string, options ...CookieOption) string
// RemoveCookie deletes a cookie by its name and path = "/".
// Tip: change the cookie's path to the current one by: RemoveCookie("name", iris.CookieCleanPath)
//
// Example: https://github.com/kataras/iris/tree/master/_examples/cookies/basic
RemoveCookie(name string, options ...CookieOption)
// VisitAllCookies accepts a visitor function which is called
// on each (request's) cookies' name and value.
VisitAllCookies(visitor func(name string, value string))

// MaxAge returns the "cache-control" request header's value
// seconds as int64
// if header not found or parse failed then it returns -1.
MaxAge() int64

// +-----+
// | Advanced: Response Recorder and Transactions |
// +-----+

// Record transforms the context's basic and direct responseWriter to a ResponseRecorder
// which can be used to reset the body, reset headers, get the body,
// get & set the status code at any time and more.
Record()
// Recorder returns the context's ResponseRecorder
// if not recording then it starts recording and returns the new context's ResponseRecorder
Recorder() *ResponseRecorder
// IsRecording returns the response recorder and a true value
// when the response writer is recording the status code, body, headers and so on,
// else returns nil and false.
IsRecording() (*ResponseRecorder, bool)

// BeginTransaction starts a scoped transaction.
//
// You can search third-party articles or books on how Business Transaction works (it's quite simple, especially here).
//
// Note that this is unique and new
// (=I have never seen any other examples or code in Golang on this subject, so far, as with the most of iris features...)
// it's not covers all paths,
// such as databases, this should be managed by the libraries you use to make your database connection,
// this transaction scope is only for context's response.
// Transactions have their own middleware ecosystem also, look iris.go:UseTransaction.
//
// See https://github.com/kataras/iris/tree/master/_examples/ for more
BeginTransaction(pipe func(t *Transaction))
// SkipTransactions if called then skip the rest of the transactions
// or all of them if called before the first transaction
SkipTransactions()
// TransactionsSkipped returns true if the transactions skipped or canceled at all.
TransactionsSkipped() bool

// Exec calls the `context/Application#ServeCtx`
// based on this context but with a changed method and path
// like it was requested by the user, but it is not.
//

```

```
// Offline means that the route is registered to the iris and have all features that a normal route has
// BUT it isn't available by browsing, its handlers executed only when other handler's context call them
// it can validate paths, has sessions, path parameters and all.
//
// You can find the Route by app.GetRoute("theRouteName")
// you can set a route name as: myRoute := app.Get("/mypath", handler)("theRouteName")
// that will set a name to the route and returns its RouteInfo instance for further usage.
//
// It doesn't changes the global state, if a route was "offline" it remains offline.
//
// app.None(...) and app.GetRoutes().Offline(route).Online(route, method)
//
// Example: https://github.com/kataras/iris/tree/master/\_examples/routing/route-state
//
// User can get the response by simple using rec := ctx.Recorder(); rec.Body()/rec.StatusCode()/rec.Header().
//
// Context's Values and the Session are kept in order to be able to communicate via the result route.
//
// It's for extreme use cases, 99% of the times will never be useful for you.
```

Exec(method, path [string](#))

```
// RouteExists reports whether a particular route exists
// It will search from the current subdomain of context's host, if not inside the root domain.
```

RouteExists(method, path [string](#)) [bool](#)

```
// Application returns the iris app instance which belongs to this context.
// Worth to notice that this function returns an interface
// of the Application, which contains methods that are safe
// to be executed at serve-time. The full app's fields
// and methods are not available here for the developer's safety.
```

Application() [Application](#)

```
// String returns the string representation of this request.
// Each context has a unique string representation.
// It can be used for simple debugging scenarios, i.e print context as string.
//
// What it returns? A number which declares the length of the
// total `String` calls per executable application, followed
// by the remote IP (the client) and finally the method:url.
```

String() [string](#)

}

API Versioning

The [versioning](#) subpackage provides [semver](#) versioning for your APIs. It implements all the suggestions written at [api-guidelines](#) and more.

The version comparison is done by the [go-version](#) package. It supports matching over patterns like `">= 1.0, < 3"` and etc.

```
import (  
    // [...]  
  
    "github.com/kataras/iris/v12"  
    "github.com/kataras/iris/v12/versioning"  
)
```

Features

- per route version matching, a normal iris handler with "switch" cases via Map for version => handler
- per group versioned routes and deprecation API
- version matching like ">= 1.0, < 2.0" or just "2.0.1" and etc.
- version not found handler (can be customized by simply adding the versioning.NotFound: customNotMatchVersionHandler on the Map)
- version is retrieved from the "Accept" and "Accept-Version" headers (can be customized via middleware)
- respond with "X-API-Version" header, if version found.
- deprecation options with customizable "X-API-Warn", "X-API-Deprecation-Date", "X-API-Deprecation-Info" headers via `Deprecated` wrapper.

Get version

Current request version is retrieved by `versioning.GetVersion(ctx)` .

By default the `GetVersion` will try to read from:

- `Accept` header, i.e `Accept: "application/json; version=1.0"`
- `Accept-Version` header, i.e `Accept-Version: "1.0"`

You can also set a custom version for a handler via a middleware by using the context's store values. For example:

```
func(ctx iris.Context) {  
    ctx.Values().Set(versioning.Key, ctx.URLParamDefault("version", "1.0"))  
    ctx.Next()  
}
```

Match version to handler

The `versioning.NewMatcher(versioning.Map) iris.Handler` creates a single handler which decides what handler need to be executed based on the requested version.

```
app := iris.New()  
  
// middleware for all versions.  
myMiddleware := func(ctx iris.Context) {  
    // [...]  
    ctx.Next()  
}  
  
myCustomNotVersionFound := func(ctx iris.Context) {  
    ctx.StatusCode(404)  
    ctx.Writef("%s version not found", versioning.GetVersion(ctx))  
}  
  
userAPI := app.Party("/api/user")  
userAPI.Get("/", myMiddleware, versioning.NewMatcher(versioning.Map{  
    "1.0":          sendHandler(v10Response),  
    ">= 2, < 3":    sendHandler(v2Response),  
    versioning.NotFound: myCustomNotVersionFound,  
})))
```

Deprecation

Using the `versioning.Deprecated(handler iris.Handler, options versioning.DeprecationOptions) iris.Handler` function you can mark a specific handler version as deprecated.

```

v10Handler := versioning.Deprecated(sendHandler(v10Response), versioning.DeprecationOptions{
    // if empty defaults to: "WARNING! You are using a deprecated version of this API."
    WarnMessage string
    DeprecationDate time.Time
    DeprecationInfo string
})

userAPI.Get("/", versioning.NewMatcher(versioning.Map{
    "1.0": v10Handler,
    // [...]
}))

```

This will make the handler to send these headers to the client:

- "X-API-Warn": options.WarnMessage
- "X-API-Deprecation-Date": context.FormatTime(ctx, options.DeprecationDate))
- "X-API-Deprecation-Info": options.DeprecationInfo

versioning.DefaultDeprecationOptions can be passed instead if you don't care about Date and Info.

Grouping routes by version

Grouping routes by version is possible as well.

Using the `versioning.NewGroup(version string) *versioning.Group` function you can create a group to register your versioned routes. The `versioning.RegisterGroups(r iris.Party, versionNotFoundHandler iris.Handler, groups ...*versioning.Group)` must be called in the end in order to register the routes to a specific `Party`.

```

app := iris.New()

userAPI := app.Party("/api/user")
// [...] static serving, middlewares and etc goes here].

userAPIV10 := versioning.NewGroup("1.0")
userAPIV10.Get("/", sendHandler(v10Response))

userAPIV2 := versioning.NewGroup(">= 2, < 3")
userAPIV2.Get("/", sendHandler(v2Response))
userAPIV2.Post("/", sendHandler(v2Response))
userAPIV2.Put("/other", sendHandler(v2Response))

versioning.RegisterGroups(userAPI, versioning.NotFoundHandler, userAPIV10, userAPIV2)

```

A middleware can be registered to the actual `iris.Party` only, using the methods we learnt above, i.e by using the `versioning.Match` in order to detect what code/handler you want to be executed when "x" or no version is requested.

Deprecation for Group

Just call the `Deprecated(versioning.DeprecationOptions)` on the group you want to notify your API consumers that this specific version is deprecated.

```

userAPIV10 := versioning.NewGroup("1.0").Deprecated(versioning.DefaultDeprecationOptions)

```

Compare version manually from inside your handlers

```

// reports if the "version" is matching to the "is".
// the "is" can be a constraint like ">= 1, < 3".
If(version string, is string) bool

```

```

// same as `If` but expects a Context to read the requested version.
Match(ctx iris.Context, expectedVersion string) bool

```



```
app.Get("/api/user", func(ctx iris.Context) {
    if versioning.Match(ctx, ">= 2.2.3") {
        // [logic for >= 2.2.3 version of your handler goes here]
        return
    }
})
```

Content Negotiation

Sometimes a server application needs to serve different representations of a resource at the same URI. Of course this can be done by hand, manually checking the `Accept` request header and push the requested form of the content. However, as your app manages more resources and different kind of representations this can be very painful, as you may need to check for `Accept-Charset` , `Accept-Encoding` , put some server-side priorities , handle the errors correctly and e.t.c.

There are some web frameworks in Go already struggle to implement a feature like this but they don't do it correctly:

- they don't handle accept-charset at all
- they don't handle accept-encoding at all
- they don't send error status code (406 not acceptable) as RFC proposes and more...

But, fortunately for us, **Iris always follows the best practises and the Web standards.**

Based on:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Charset>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Encoding>

Implemented on:

- <https://github.com/kataras/iris/pull/1316/commits/8ee0de51c593fe0483fbea38117c3c88e065f2ef>

Example

```
type testdata struct {
    Name string `json:"name" xml:"Name"`
    Age  int    `json:"age" xml:"Age"`
}
```

Render a resource with "gzip" encoding algorithm as application/json or text/xml or application/xml

- when client's accept header contains one of them
- or JSON (the first declared) if accept is empty,
- and when client's accept-encoding header contains "gzip" or it's empty.

```
app.Get("/resource", func(ctx iris.Context) {
    data := testdata{
        Name: "test name",
        Age:  26,
    }

    ctx.Negotiation().JSON().XML().EncodingGzip()

    _, err := ctx.Negotiate(data)
    if err != nil {
        ctx.Writef("%v", err)
    }
})
```

OR define them in a middleware and call Negotiate with nil in the final handler.

```
ctx.Negotiation().JSON(data).XML(data).Any("content for */*")
ctx.Negotiate(nil)
```

```
app.Get("/resource2", func(ctx iris.Context) {
    jsonAndXML := testdata{
        Name: "test name",
        Age:  26,
    }

    ctx.Negotiation().
        JSON(jsonAndXML).
        XML(jsonAndXML).
        HTML("<h1>Test Name</h1><h2>Age 26</h2>")

    ctx.Negotiate(nil)
})
```

[Read the full example.](#)

Documentation

The [Context.Negotiation](#) method creates once and returns the negotiation builder to build server-side available prioritized content for specific content type(s), charset(s) and encoding algorithm(s).

```
Context.Negotiation() *context.NegotiationBuilder
```

The [Context.Negotiate](#) method used for serving different representations of a resource at the same URI. It returns `context.ErrContentNotSupported` when not matched mime type(s).

- The "v" can be a single [iris.N](#) struct value.
- The "v" can be any value completes the [context.ContentSelector](#) interface.
- The "v" can be any value completes the [context.ContentNegotiator](#) interface.
- The "v" can be any value of struct(JSON, JSONP, XML, YAML) or string(TEXT, HTML) or []byte(Markdown, Binary) or []byte with any matched mime type.
- If the "v" is nil, the `Context.Negotitation()` builder's content will be used instead, otherwise "v" overrides builder's content (server mime types are still retrieved by its registered, supported, mime list)
- Set mime type priorities by [Negotiation\(\).MIME.Text.JSON.XML.HTML....](#)
- Set charset priorities by [Negotiation\(\).Charset\(...\)](#).
- Set encoding algorithm priorities by [Negotiation\(\).Encoding\(...\)](#).
- Modify the accepted by [Negotiation\(\).Accept/Override\(\)/XML\(\).JSON\(\).Charset\(...\).Encoding\(...\)....](#)

```
Context.Negotiate(v interface{}) (int, error)
```

Response Recorder

A response recorder is an Iris specific `http.ResponseWriter` which records the sent body, status code and headers that you can manipulate at any handler inside a route's request handlers chain.

- 1. Call `Context.Record()` before send data.
- 2. The `Context.Recorder()` returns a [ResponseRecorder](#). Its methods can be used to manipulate or retrieve the response.

The `ResponseRecorder` type contains the standard Iris `ResponseWriter` methods plus the following methods.

Body **returns** the body tracked from the writer so far. Do not use this for edit.

```
Body() []byte
```

Use this to clear the body.

```
ResetBody()
```

Use `Write/Writef/WriteString` to stream write and `SetBody/SetBodyString` to **set** body instead.

```
Write(contents []byte) (int, error)

Writef(format string, a ...interface{}) (n int, err error)

WriteString(s string) (n int, err error)

SetBody(b []byte)

SetBodyString(s string)
```

Reset headers to their original state, before `Context.Record` call.

```
ResetHeaders()
```

Clear all headers.

```
ClearHeaders()
```

Reset resets the response body, headers and the status code header.

```
Reset()
```

Example

Record operation log in global Interceptor.

```
package main

import "github.com/kataras/iris/v12"

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

    // start record.
    app.Use(func(ctx iris.Context) {
        ctx.Record()
        ctx.Next()
    })

    // collect and "log".
    app.Done(func(ctx iris.Context) {
        body := ctx.Recorder().Body()

        // Should print success.
        app.Logger().Infof("sent: %s", string(body))
    })
}
```

Register the routes...

```
app.Get("/save", func(ctx iris.Context) {
    ctx.WriteString("success")
    ctx.Next() // calls the Done middleware(s).
})
```

Or to remove the need of `ctx.Next` in your main handlers, modify the Iris handlers [execution rules](#) as follows.

```
// It applies per Party and its children,
// therefore, you can create a routes := app.Party("/path")
// and set middlewares, their rules and the routes there as well.
app.SetExecutionRules(iris.ExecutionRules{
    Done: iris.ExecutionOptions{Force: true},
})

// [The routes...]
app.Get("/save", func(ctx iris.Context) {
    ctx.WriteString("success")
})
```

In addition to that, Iris provides a comprehensive API for **Transactions**. Learn more about it by running an [example](#).

HTTP Referer

The HTTP referer(originally a misspelling of **referrer**) is an optional HTTP header field that identifies the address of the webpage (i.e. the URI or IRI) which is linked to the resource being requested. By checking the referrer, the new webpage can see where the request originated.

Read more at [wikipedia](#)

Iris uses the [Shopify's gorereferrer](#) package to expose the `Context.GetReferrer()` method.

The `GetReferrer` Context's method extracts and returns the information from the `"Referer"` header as specified in <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy> or by the URL query parameter `"referrer"` .

`GetReferrer()` Referrer

Which `Referrer` looks like that:

```
type (
  Referrer struct {
    Type      ReferrerType
    Label      string
    URL        string
    Subdomain  string
    Domain     string
    Tld        string
    Path       string
    Query      string
    GoogleType ReferrerGoogleSearchType
  }
)
```

The `ReferrerType` is the enum for a Referrer.Type value (indirect, direct, email, search, social). The available types are:

ReferrerInvalid
ReferrerIndirect
ReferrerDirect
ReferrerEmail
ReferrerSearch
ReferrerSocial

The `GoogleType` can be one of those:

ReferrerNotGoogleSearch
ReferrerGoogleOrganicSearch
ReferrerGoogleAdwords

Example

```
package main

import "github.com/kataras/iris/v12"

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

    app.Get("/", func(ctx iris.Context) {
        r := ctx.GetReferrer()
        switch r.Type {
        case iris.ReferrerSearch:
            ctx.Writef("Search %s: %s\n", r.Label, r.Query)
            ctx.Writef("Google: %s\n", r.GoogleType)
        case iris.ReferrerSocial:
            ctx.Writef("Social %s\n", r.Label)
        case iris.ReferrerIndirect:
            ctx.Writef("Indirect: %s\n", r.URL)
        }
    })

    app.Run(iris.Addr(":8080"))
}
```

How to `curl` :

```
curl http://localhost:8080?\
referer=https://twitter.com/Xinterio/status/1023566830974251008

curl http://localhost:8080?\
referer=https://www.google.com/search?q=Top+6+golang+web+frameworks\
&oq=Top+6+golang+web+frameworks
```

Request Authentication

Iris offers request authentication through its [jwt middleware](#). In this chapter you will learn the basics of how to use JWT with Iris.

1. Install it by executing the following shell command:

```
$ go get github.com/iris-contrib/middleware/jwt
```

2. To create a new jwt middleware use the `jwt.New` function. This example extracts the token through a `"token"` url parameter. Authenticated clients should be designed to set that with a signed token. The default jwt middleware's behavior to extract a token value is by the `Authentication: Bearer $TOKEN` header.

The jwt middleware has three methods to validate tokens.

- The first one is the `Serve` method - it is an `iris.Handler` ,
- the second one is the `CheckJWT(iris.Context) bool` and
- the third one is a helper to retrieve the validated token - the `Get(iris.Context) *jwt.Token` .

3. To register it you simply prepend the jwt `j.Serve` middleware to a specific group of routes, a single route, or globally e.g. `app.Get("/secured", j.Serve, myAuthenticatedHandler)` .

4. To generate a token inside a handler that accepts a user payload and responses the signed token which later on can be sent through client requests headers or url parameter in this case, e.g. `jwt.NewToken` or `jwt.NewTokenWithClaims` .

Example


```

import (
    "github.com/kataras/iris/v12"
    "github.com/iris-contrib/middleware/jwt"
)

func getTokenHandler(ctx iris.Context) {
    token := jwt.NewTokenWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
        "foo": "bar",
    })

    // Sign and get the complete encoded token as a string using the secret
    tokenString, _ := token.SignedString([]byte("My Secret"))

    ctx.HTML(`Token: ` + tokenString + `  
<br/>`
    <a href="/secured?token=` + tokenString + `"/secured?token=` + tokenString + `</a>`)
}

func myAuthenticatedHandler(ctx iris.Context) {
    user := ctx.Values().Get("jwt").(*jwt.Token)

    ctx.Writef("This is an authenticated request\n")
    ctx.Writef("Claim content:\n")

    foobar := user.Claims.(jwt.MapClaims)
    for key, value := range foobar {
        ctx.Writef("%s = %s", key, value)
    }
}

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

    j := jwt.New(jwt.Config{
        // Extract by "token" url parameter.
        Extractor: jwt.FromParameter("token"),

        ValidationKeyGetter: func(token *jwt.Token) (interface{}, error) {
            return []byte("My Secret"), nil
        },
        SigningMethod: jwt.SigningMethodHS256,
    })

    app.Get("/", getTokenHandler)
    app.Get("/secured", j.Serve, myAuthenticatedHandler)
    app.Run(iris.Addr(":8080"))
}

```

You can use any payload as "claims".

URL Query Parameters

The Iris Context has two methods that return the `net/http` standard `http.ResponseWriter` and `http.Request` values as we already mention at the previous chapters.

- `Context.Request()`
- `Context.ResponseWriter()`

However, except the unique Iris features and helpers that Iris Context offers, for easier development, we provide some wrappers of existing `net/http` capabilities as well.

This is the full list of methods that can help you when working with URL query string.

```
// URLParam returns true if the url parameter exists, otherwise false.
URLParamExists(name string) bool
// URLParamDefault returns the get parameter from a request,
// if not found then "def" is returned.
URLParamDefault(name string, def string) string
// URLParam returns the get parameter from a request, if any.
URLParam(name string) string
// URLParamTrim returns the url query parameter with
// trailing white spaces removed from a request.
URLParamTrim(name string) string
// URLParamTrim returns the escaped url query parameter from a request.
URLParamEscape(name string) string
// URLParamInt returns the url query parameter as int value from a request,
// returns -1 and an error if parse failed.
URLParamInt(name string) (int, error)
// URLParamIntDefault returns the url query parameter as int value from a request,
// if not found or parse failed then "def" is returned.
URLParamIntDefault(name string, def int) int
// URLParamInt32Default returns the url query parameter as int32 value from a request,
// if not found or parse failed then "def" is returned.
URLParamInt32Default(name string, def int32) int32
// URLParamInt64 returns the url query parameter as int64 value from a request,
// returns -1 and an error if parse failed.
URLParamInt64(name string) (int64, error)
// URLParamInt64Default returns the url query parameter as int64 value from a request,
// if not found or parse failed then "def" is returned.
URLParamInt64Default(name string, def int64) int64
// URLParamFloat64 returns the url query parameter as float64 value from a request,
// returns -1 and an error if parse failed.
URLParamFloat64(name string) (float64, error)
// URLParamFloat64Default returns the url query parameter as float64 value from a request,
// if not found or parse failed then "def" is returned.
URLParamFloat64Default(name string, def float64) float64
// URLParamBool returns the url query parameter as boolean value from a request,
// returns an error if parse failed or not found.
URLParamBool(name string) (bool, error)
// URLParams returns a map of GET query parameters separated by comma if more than one
// it returns an empty map if nothing found.
URLParams() map[string]string
```

Query string parameters are parsed using the existing underlying request object. The request responds to a url matching: `/welcome?firstname=Jane&lastname=Doe`.

- `ctx.URLParam("lastname") == ctx.Request().URL.Query().Get("lastname")`

Example Code:

```
app.Get("/welcome", func(ctx iris.Context) {
    firstname := ctx.URLParamDefault("firstname", "Guest")
    lastname := ctx.URLParam("lastname")

    ctx.Writef("Hello %s %s", firstname, lastname)
})
```

Forms

Form, post data and uploaded files can be retrieved using the following Context's methods.

```
// FormValueDefault returns a single parsed form value by its "name",
// including both the URL field's query parameters and the POST or PUT form data.
//
// Returns the "def" if not found.
FormValueDefault(name string, def string) string
// FormValue returns a single parsed form value by its "name",
// including both the URL field's query parameters and the POST or PUT form data.
FormValue(name string) string
// FormValues returns the parsed form data, including both the URL
// field's query parameters and the POST or PUT form data.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris.WithPostMaxMemory` configurator at
// main configuration passed on `app.Run`'s second argument.
//
// NOTE: A check for nil is necessary.
FormValues() map[string][]string
```

```
// PostValueDefault returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name".
//
// If not found then "def" is returned instead.
PostValueDefault(name string, def string) string
// PostValue returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name"
PostValue(name string) string
// PostValueTrim returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", without trailing spaces.
PostValueTrim(name string) string
// PostValueInt returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as int.
//
// If not found returns -1 and a non-nil error.
PostValueInt(name string) (int, error)
// PostValueIntDefault returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as int.
//
// If not found returns or parse errors the "def".
PostValueIntDefault(name string, def int) int
// PostValueInt64 returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as float64.
//
// If not found returns -1 and a no-nil error.
PostValueInt64(name string) (int64, error)
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as int64.
//
// If not found or parse errors returns the "def".
PostValueInt64Default(name string, def int64) int64
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as float64.
//
// If not found returns -1 and a non-nil error.
PostValueFloat64(name string) (float64, error)
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as float64.
//
// If not found or parse errors returns the "def".
PostValueFloat64Default(name string, def float64) float64
// PostValueInt64Default returns the parsed form data from POST, PATCH,
// or PUT body parameters based on a "name", as bool.
//
// If not found or value is false, then it returns false, otherwise true.
PostValueBool(name string) (bool, error)
// PostValues returns all the parsed form data from POST, PATCH,
```

```
// or PUT body parameters based on a "name" as a string slice.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris.WithPostMaxMemory` configurator at
// main configuration passed on `app.Run`'s second argument.
PostValues(name string) []string
// FormFile returns the first uploaded file that received from the client.
//
// The default form's memory maximum size is 32MB, it can be changed by the
// `iris.WithPostMaxMemory` configurator at
// main configuration passed on `app.Run`'s second argument.
FormFile(key string) (multipart.File, *multipart.FileHeader, error)
```

Multipart/Urlencoded Form

```
func main() {
    app := iris.Default()

    app.Post("/form_post", func(ctx iris.Context) {
        message := ctx.FormValue("message")
        nick := ctx.FormValueDefault("nick", "anonymous")

        ctx.JSON(iris.Map{
            "status": "posted",
            "message": message,
            "nick":    nick,
        })
    })

    app.Run(iris.Addr(":8080"))
}
```

Another example: query + post form

```
POST /post?id=1234&page=1 HTTP/1.1
Content-Type: application/x-www-form-urlencoded
```

```
name=manu&message=this_is_great
```

```
func main() {
    app := iris.Default()

    app.Post("/post", func(ctx iris.Context) {
        id := ctx.URLParam("id")
        page := ctx.URLParamDefault("page", "0")
        name := ctx.FormValue("name")
        message := ctx.FormValue("message")
        // or `ctx.PostValue` for POST, PUT & PATCH-only HTTP Methods.

        app.Logger().Infof("id: %s; page: %s; name: %s; message: %s",
            id, page, name, message)
    })

    app.Run(iris.Addr(":8080"))
}
```

```
id: 1234; page: 1; name: manu; message: this_is_great
```

Upload files

Iris Context offers a helper for uploading files (saving files to host system's hard disk from request file data). Read more about the `Context.UploadFormFiles` method below.

UploadFormFiles uploads any received file(s) from the client to the system physical location "destDirectory".

The second optional argument "before" gives caller the chance to modify the *multipart.FileHeader before saving to the disk, it can be used to change a file's name based on the current request, all FileHeader's options can be changed. You can ignore it if you don't need to use this capability before saving a file to the disk.

Note that it doesn't check if request body streamed.

Returns the copied length as int64 and a not nil error if at least one new file can't be created due to the operating system's permissions or `net/http.ErrMissingFile` if no file received.

If you want to receive & accept files and manage them manually you can use the `Context.FormFile` instead and create a copy function that suits your needs, the below is for generic usage.

The default form's memory maximum size is 32MB, it can be changed by the `iris#WithPostMaxMemory` configurator at main configuration passed on `app.Run` 's second argument.

```
UploadFormFiles(destDirectory string,
                before ...func(Context, *multipart.FileHeader)) (n int64, err error)
```

Example Code:

```
const maxSize = 5 << 20 // 5MB

func main() {
    app := iris.Default()
    app.Post("/upload", iris.LimitRequestBodySize(maxSize), func(ctx iris.Context) {
        //
        // UploadFormFiles
        // uploads any number of incoming files ("multiple" property on the form input).
        //

        // The second, optional, argument
        // can be used to change a file's name based on the request,
        // at this example we will showcase how to use it
        // by prefixing the uploaded file with the current user's ip.
        ctx.UploadFormFiles("./uploads", beforeSave)
    })

    app.Run(iris.Addr(":8080"))
}

func beforeSave(ctx iris.Context, file *multipart.FileHeader) {
    ip := ctx.RemoteAddr()
    // make sure you format the ip in a way
    // that can be used for a file name (simple case):
    ip = strings.Replace(ip, ".", "_", -1)
    ip = strings.Replace(ip, ":", "_", -1)

    // you can use the time.Now, to prefix or suffix the files
    // based on the current time as well, as an exercise.
    // i.e unixTime :=    time.Now().Unix()
    // prefix the Filename with the $IP-
    // no need for more actions, internal uploader will use this
    // name to save the file into the "./uploads" folder.
    file.Filename = ip + "-" + file.Filename
}
```

How to `curl` :

```
curl -X POST http://localhost:8080/upload \
-F "files[]=@./myfile.zip" \
-F "files[]=@./mysecondfile.zip" \
-H "Content-Type: multipart/form-data"
```

More examples can be found at https://github.com/kataras/iris/tree/master/_examples/http_request.

Model Validation

Iris does not contain built-in method for validating request data such as "Models". However, you are not limited. In this example we will learn how to use the [go-playground/validator.v9](#) for request values validation.

Check the full docs on tags usage [here](#).

```
$ go get gopkg.in/go-playground/validator.v9
```

Note that you need to set the corresponding binding tag on all fields you want to bind. For example, when binding from JSON, set `json:"fieldname"` .


```

package main

import (
    "fmt"

    "github.com/kataras/iris/v12"

    "gopkg.in/go-playground/validator.v9"
)

// User contains user information.
type User struct {
    FirstName    string    `json:"fname"`
    LastName     string    `json:"lname"`
    Age          uint8     `json:"age" validate:"gte=0,lte=130"`
    Email        string    `json:"email" validate:"required,email"`
    FavouriteColor string    `json:"favColor" validate:"hexcolor|rgb|rgba"`
    Addresses    []*Address `json:"addresses" validate:"required,dive,required"`
}

// Address houses a users address information.
type Address struct {
    Street string `json:"street" validate:"required"`
    City   string `json:"city" validate:"required"`
    Planet string `json:"planet" validate:"required"`
    Phone  string `json:"phone" validate:"required"`
}

// Use a single instance of Validate, it caches struct info.
var validate *validator.Validate

func main() {
    validate = validator.New()

    // Register validation for 'User'
    // NOTE: only have to register a non-pointer type for 'User', validator
    // internally dereferences during it's type checks.
    validate.RegisterStructValidation(UserStructLevelValidation, User{})

    app := iris.New()
    app.Post("/user", func(ctx iris.Context) {
        var user User
        if err := ctx.ReadJSON(&user); err != nil {
            // [handle error...]
        }

        // Returns InvalidValidationError for bad validation input,
        // nil or ValidationErrors ( []FieldError )
        err := validate.Struct(user)
        if err != nil {

            // This check is only needed when your code could produce
            // an invalid value for validation such as interface with nil
            // value most including myself do not usually have code like this.
            if _, ok := err.(*validator.InvalidValidationError); ok {
                ctx.StatusCode(iris.StatusInternalServerError)
                ctx.WriteString(err.Error())
                return
            }

            ctx.StatusCode(iris.StatusBadRequest)
            for _, err := range err.(validator.ValidationErrors) {
                fmt.Println()
                fmt.Println(err.Namespace())
                fmt.Println(err.Field())
            }
        }
    })
}

```



```

        fmt.Println(err.StructNamespace())
        fmt.Println(err.StructField())
        fmt.Println(err.Tag())
        fmt.Println(err.ActualTag())
        fmt.Println(err.Kind())
        fmt.Println(err.Type())
        fmt.Println(err.Value())
        fmt.Println(err.Param())
        fmt.Println()
    }

    return
}

// [save user to database...]
})

app.Run(iris.Addr(":8080"))
}

func UserStructLevelValidation(sl validator.StructLevel) {
    user := sl.Current().Interface().(User)

    if len(user.FirstName) == 0 && len(user.LastName) == 0 {
        sl.ReportError(user.FirstName, "FirstName", "fname", "fnameorlname", "")
        sl.ReportError(user.LastName, "LastName", "lname", "fnameorlname", "")
    }
}

```

Example request of JSON form:

```

{
  "fname": "",
  "lname": "",
  "age": 45,
  "email": "mail@example.com",
  "favColor": "#000",
  "addresses": [{
    "street": "Eavesdown Docks",
    "planet": "Persphone",
    "phone": "none",
    "city": "Unknown"
  }]
}

```

Example can be found at: https://github.com/kataras/iris/tree/master/_examples/http_request/read-json-struct-validation/main.go.

Cache

Sometimes is important to cache routes that serve static content to make your web app performs faster and not spend time on re-building a response for clients.

There are two ways to achieve HTTP caching. One is to store the contents per handler on the server-side and the other is by checking some headers and send `304 not modified` in order to let the browser or any compatible client to handle the caching by itself.

Iris provides both server and client caching through its [iris/cache](#) subpackage which exports the `iris.Cache` and `iris.Cache304` middlewares.

Let's see their outline.

Cache

`Cache` is a middleware that provides server-side cache functionality to the next handlers, can be used as: `app.Get("/", iris.Cache, aboutHandler)` .

`Cache` accepts one single parameter: the cache expiration duration. If the expiration is invalid value, <=2 seconds, then expiration is taken by the "cache-control's maxage" header

All types of response can be cached, templates, json, text, anything.

Use it for server-side caching, see the `Cache304` for an alternative approach that may be more suited to your needs.

```
func Cache(expiration time.Duration) Handler
```

For more options and customization use the [kataras/iris/cache.Cache](#) which returns a [structure from which you can add or remove "Rules"](#) instead.

NoCache

`NoCache` is a middleware which overrides the Cache-Control, Pragma and Expires headers in order to disable the cache during the browser's back and forward feature.

A good use of this middleware is on HTML routes; to refresh the page even on "back" and "forward" browser's arrow buttons.

```
func NoCache(Context)
```

See `StaticCache` for the opposite behavior.

StaticCache

`StaticCache` returns a middleware for caching static files by sending the "Cache-Control" and "Expires" headers to the client. It accepts a single input parameter, the "cacheDur", a time.Duration that it's used to calculate the expiration.

If "cacheDur" <=0 then it returns the `NoCache` middleware instaed to disable the caching between browser's "back" and "forward" actions.

Usage: `app.Use(iris.StaticCache(24 * time.Hour))` or `app.Use(iris.StaticCache(-1))` .

```
func StaticCache(cacheDur time.Duration)Handler
```

A middleware, which is a simple Handler, can be called inside another handler as well, example:

```
cacheMiddleware := iris.StaticCache(...)

func(ctx iris.Context){
    cacheMiddleware(ctx)
    [...]
}
```

Cache304

`Cache304` returns a middleware sends a `StatusNotModified` (304) whenever the "If-Modified-Since" request header (time) is before the time.Now() + expiresEvery (always compared to their UTC values).

Clients that are compatible with the HTTP RCF (all browsers are and tools like postman) will correctly handle the caching.

The only disadvantage of using that instead of server-side caching is that this method will send a 304 status code instead of 200, So, if you use it side by side with other micro services you have to check for that status code as well for a valid response.

Developers are free to extend this method's behavior by watching system directories changes manually and use of the `ctx.WriteWithExpiration` with a "modtime" based on the file modified date, similar to the `HandleDir` (which sends status OK(200) and browser disk caching instead of 304).

```
func Cache304(expiresEvery time.Duration) Handler
```

Examples can be found at: https://github.com/kataras/iris/tree/master/_examples/cache.

File Server

Serve static files from a specific directory (system physical or embedded to the application) is done by the `Party.HandleDir` method.

`HandleDir` registers a handler that serves HTTP requests with the contents of a file system (physical or embedded).

- First parameter : the route path
- Second parameter : the system or the embedded directory that needs to be served
- Third parameter : not required, the directory options, set fields is optional.

Returns the GET *Route.

```
HandleDir(requestPath, directory string, opts ...DirOptions) (getRoute *Route)
```

The `DirOptions` structure looks like this:

```
type DirOptions struct {
    // Defaults to "/index.html", if request path is ending with **/*/$IndexName
    // then it redirects to **/*(/) which another handler is handling it,
    // that another handler, called index handler, is auto-registered by the framework
    // if end developer wasn't managed to handle it manually/by hand.
    IndexName string
    // Should files served under gzip compression?
    Gzip bool

    // List the files inside the current requested directory if `IndexName` not found.
    ShowList bool
    // If `ShowList` is true then this function will be used instead
    // of the default one to show the list of files of a current requested directory(dir).
    DirList func(ctx iris.Context, dirName string, dir http.File) error

    // When embedded.
    Asset      func(name string) ([]byte, error)
    AssetInfo  func(name string) (os.FileInfo, error)
    AssetNames func() []string

    // Optional validator that loops through each found requested resource.
    AssetValidator func(ctx iris.Context, name string) bool
}
```

Let's say that you have an `./assets` folder near to your executable and you want the files to be served through `http://localhost:8080/static/**/*` route.

```
app := iris.New()

app.HandleDir("/static", "./assets")

app.Run(iris.Addr(":8080"))
```

Now, if you want to embed the static files to be lived inside the executable build in order to not depend on a system directory you can use a tool like [go-bindata](#) to convert the files into `[]byte` inside your program. Let's take a quick tutorial on this and how Iris helps to serve those data.

Install go-bindata:

```
go get -u github.com/go-bindata/go-bindata/...
```

Navigate to your program directory, that the `./assets` subdirectory exists and execute:

```
$ go-bindata ./assets/...
```

The above creates a generated go file which contains three main functions: `Asset` , `AssetInfo` and `AssetNames` . Use them on the `DirOptions` :

```
// [app := iris.New...]

app.HandleDir("/static", "./assets", iris.DirOptions {
    Asset: Asset,
    AssetInfo: AssetInfo,
    AssetNames: AssetNames,
    Gzip: false,
})
```

Build your app:

```
$ go build
```

The `HandleDir` supports all the standards, including content-range, for both physical and embedded directories.

However, if you just need a handler to work with, without register a route, you can use the `iris.FileServer` package-level function instead.

The `FileServer` function returns a Handler which serves files from a specific system, physical, directory or an embedded one.

- First parameter: is the directory.
- Second parameter: optional parameter is any optional settings that the caller can use.

```
iris.FileServer(directory string, options ...DirOptions)
```

Usage

```
handler := iris.FileServer("./assets", iris.DirOptions {  
    ShowList: true, Gzip: true, IndexName: "index.html",  
})
```

Examples can be found at: https://github.com/kataras/iris/tree/master/_examples/file-server

View

Iris offers support for **6 template parsers out of the box** through its universal [View Engine](#). Of course developers can still use various go template parsers as the `Context.ResponseWriter()` completes the `http.ResponseWriter` and `io.Writer` .

Iris puts some common rules and features that their original parsers don't support by default. For example we have support for `yield` , `render` , `render_r` , `current` , `urlpath` template funcs and `Layouts` and `binding` across middlewares and **embedded template files** for all Engines.

To use unique features of a template engine you have to learn the characteristics and the syntax by reading its documentation (click the links below). Choose what fits your app's needs the most.

Let's see the list of the built-in view engines:

Engine	Declaration	Underline Template Parser
std template/html	<code>iris.HTML(...)</code>	html/template package
django	<code>iris.Django(...)</code>	flosch/pongo2 package
handlebars	<code>iris.Handlebars(...)</code>	Joker/jade package
amber	<code>iris.Amber(...)</code>	aymerick/raymond package
pug(jade)	<code>iris.Pug(...)</code>	eknkc/amber package
jet	<code>iris.Jet(...)</code>	CloudyKit/jet package

One or more view engines can be registered in the same Application. To **register** a view engine use the `RegisterView(ViewEngine)` method.

Load all templates from the `"/views"` folder where extension is `".html"` and parse them using the standard `html/template` package.

```
// [app := iris.New...]
tpl := iris.HTML("./views", ".html")
app.RegisterView(tpl)
```

To **render or execute** a view use the `Context.View` method inside the main route's handler.

```
ctx.View("hi.html")
```

To **bind** Go values with key-value pattern inside a view through middleware or main handler use the `Context.ViewData` method before the `Context.View` one.

Bind: `{{.message}}` with `"Hello world!"` .

```
ctx.ViewData("message", "Hello world!")
```

To bind a Go **model** to a view you have two options:

1.

```
ctx.ViewData("user", User{})

// variable binding as {{.user.Name}}
```

2.

```
ctx.View("user-page.html", User{})

// root binding as {{.Name}}
```

To **add a template function** use the `AddFunc` method of the preferred view engine.

```
//      func name, input arguments, render value
tpl.AddFunc("greet", func(s string) string {
    return "Greetings " + s + "!"
})
```

To **reload on local file changes** call the view engine's `Reload` method.

```
tpl.Reload(true)
```

To use **embedded** templates and not depend on local file system use the [go-bindata](#) external tool and pass its `Asset` and `AssetNames` functions to the `Binary` method of the preferred view engine.

```
tpl.Binary(Asset, AssetNames)
```

Example Code:

Please read the *comments* too.

```
// file: main.go
package main

import "github.com/kataras/iris/v12"

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

    // Parse all templates from the "./views" folder
    // where extension is ".html" and parse them
    // using the standard `html/template` package.
    tpl := iris.HTML("./views", ".html")

    // Enable re-build on local template files changes.
    tpl.Reload(true)

    // Default template funcs are:
    //
    // - {{ urlpath "myNamedRoute" "pathParameter_ifNeeded" }}
    // - {{ render "header.html" }}
    // and partial relative path to current page:
    // - {{ render_r "header.html" }}
    // - {{ yield }}
    // - {{ current }}
    // Register a custom template func:
    tpl.AddFunc("greet", func(s string) string {
        return "Greetings " + s + "!"
    })

    // Register the view engine to the views,
    // this will load the templates.
    app.RegisterView(tpl)

    // Method:    GET
    // Resource:  http://localhost:8080
    app.Get("/", func(ctx iris.Context) {
        // Bind: {{.message}} with "Hello world!"
        ctx.ViewData("message", "Hello world!")
        // Render template file: ./views/hi.html
        ctx.View("hi.html")
    })

    app.Run(iris.Addr(":8080"))
}
```

```
<!-- file: ./views/hi.html -->
<html>
<head>
    <title>Hi Page</title>
</head>
<body>
    <h1>{{.message}}</h1>
    <strong>{{greet "to you"}}</strong>
</body>
</html>
```

Open a browser tab at <http://localhost:8080>.

The **rendered result** will look like this:

```
<html>
<head>
  <title>Hi Page</title>
</head>
<body>
  <h1>Hello world!</h1>
  <strong>Greetings to you!</strong>
</body>
</html>
```


Cookies

Cookies are accessible through the Request instance of Context. The `ctx.Request()` returns a `net/http#Request` .

However the Iris `Context` provides some helpers to make the most common use cases of cookies easier accessible to you and without any custom additional code of yours that would be required if you just using the Request's Cookies methods.

Set a Cookie

The `SetCookie` method adds a cookie.

Use of the "options" is not required, they can be used to modify the "cookie". You'll see later on what the available options are, custom ones can be added depending on your web application requirements, this also helps to not repeat yourself in a codebase.

```
SetCookie(cookie *http.Cookie, options ...CookieOption)
```

If you want you can also use the `SetCookieKV` method which does not require an import of the `net/http` package.

```
SetCookieKV(name, value string, options ...CookieOption)
```

Note that the default expiration for a cookie set by `SetCookieKV` is 365 days. You can either use the `CookieExpires` Cookie Option(see below) or globally by setting the `kataras/iris/Context.SetCookieKVExpiration` package-level variable.

The `CookieOption` is just a type for `func(*http.Cookie)` .

Set Path

```
CookiePath(path string) CookieOption
```

Set Expiration

```
iris.CookieExpires(durFromNow time.Duration) CookieOption
```

HttpOnly

```
iris.CookieHTTPOnly(httpOnly bool) CookieOption
```

`HttpOnly` field defaults to true for `RemoveCookie` and `SetCookieKV` .

And let's go a bit further with cookie encoding.

Encode

Provides encoding functionality when adding a cookie.

Accepts a `CookieEncoder` and sets the cookie's value to the encoded value.

Users of that is the `SetCookie` and `SetCookieKV` .

```
iris.CookieEncode(encode CookieEncoder) CookieOption
```

Decode

Provides decoding functionality when retrieving a cookie.

Accepts a `CookieDecoder` and sets the cookie's value to the decoded value before return by the `GetCookie` .

User of that is the `GetCookie` .

```
iris.CookieDecode(decode CookieDecoder) CookieOption
```

Where `CookieEncoder` can be described as:

A `CookieEncoder` should encode the cookie value.

- Should accept the cookie's name as its first argument and
- as second argument the cookie value ptr.
- Should return an encoded value or an empty one if encode operation failed.
- Should return an error if encode operation failed.

```
CookieEncoder func(cookieName string, value interface{}) (string, error)
```

And `CookieDecoder` :

`CookieDecoder` should decode the cookie value.

- Should accept the cookie's name as its first argument,
- as second argument the encoded cookie value and as third argument the decoded value ptr.
- Should return a decoded value or an empty one if decode operation failed.
- Should return an error if decode operation failed.

```
CookieDecoder func(cookieName string, cookieValue string, v interface{}) error
```

Errors are not printed, so you have to know what you're doing, and remember: if you use AES it only supports key sizes of 16, 24 or 32 bytes.

You either need to provide exactly that amount or you derive the key from what you type in.

Get a Cookie

The `GetCookie` returns cookie's value by its name, returns empty string if nothing was found.

```
GetCookie(name string, options ...CookieOption) string
```

If you want more than the value then use the following instead:

```
cookie, err := ctx.Request().Cookie("name")
```

Get all Cookies

The `ctx.Request().Cookies()` method returns a slice of all the available request cookies. Sometimes you want to modify them or perform an action on each one of them, the easiest way to do that is by the `VisitAllCookies` method.

```
VisitAllCookies(visitor func(name string, value string))
```

Remove a Cookie

The `RemoveCookie` method deletes a cookie by its name and path = "/", the root one.

Tip: change the cookie's path to the current one by providing the `iris.CookieCleanPath` Cookie Options, as : `RemoveCookie("nname", iris.CookieCleanPath)` .

Also, note that the default behavior is to set its `HttpOnly` to true. It performs a removal of cookie based on the web standards.

```
RemoveCookie(name string, options ...CookieOption)
```

Sessions

When you work with an application, you open it, do some changes, and then you close it. This is much like a Session. The computer knows who you are. It knows when you start the application and when you end. But on the internet there is one problem: the web server does not know who you are or what you do, because the HTTP address doesn't maintain state.

Session variables solve this problem by storing user information to be used across multiple pages (e.g. username, favorite color, etc). By default, session variables last until the user closes the browser.

So; Session variables hold information about one single user, and are available to all pages in one application.

Tip: If you need a permanent storage, you may want to store the data in a [database](#).

Iris has its own session implementation and sessions manager live inside the [iris/sessions](#) subpackage. You'll need to import this package in order to work with HTTP Sessions.

A session is started with the `Start` function of the `Sessions` object which is created with the `New` package-level function. That function will return a `Session`.

Session variables are set with the `Session.Set` method and retrieved by the `Session.Get` and its related methods. To delete a single variable use the `Session.Delete`. To delete and invalidate the whole session use the `Session.Destroy` method.

► See all methods

Example

In this example we will only allow authenticated users to view our secret message on the `/secret` age. To get access to it, the will first have to visit `/login` to get a valid session cookie, hich logs him in. Additionally he can visit `/logout` to revoke his access to our secret message.

```
// sessions.go
package main

import (
    "github.com/kataras/iris/v12"

    "github.com/kataras/iris/v12/sessions"
)

var (
    cookieNameForSessionID = "mycookiesessionnameid"
    sess                    = sessions.New(sessions.Config{Cookie: cookieNameForSessionID})
)

func secret(ctx iris.Context) {
    // Check if user is authenticated
    if auth, _ := sess.Start(ctx).GetBoolean("authenticated"); !auth {
        ctx.StatusCode(iris.StatusForbidden)
        return
    }

    // Print secret message
    ctx.WriteString("The cake is a lie!")
}

func login(ctx iris.Context) {
    session := sess.Start(ctx)

    // Authentication goes here
    // ...

    // Set user as authenticated
    session.Set("authenticated", true)
}

func logout(ctx iris.Context) {
    session := sess.Start(ctx)

    // Revoke users authentication
    session.Set("authenticated", false)
    // Or to remove the variable:
    session.Delete("authenticated")
    // Or destroy the whole session:
    session.Destroy()
}

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

    app.Get("/secret", secret)
    app.Get("/login", login)
    app.Get("/logout", logout)

    app.Run(iris.Addr(":8080"))
}
```

```
$ go run sessions.go

$ curl -s http://localhost:8080/secret
Forbidden

$ curl -s -I http://localhost:8080/login
Set-Cookie: mysessionid=MTQ4NzE5Mz...

$ curl -s --cookie "mysessionid=MTQ4NzE5Mz..." http://localhost:8080/secret
The cake is a lie!
```

Middlware

When you want to use the `Session` in the same request handler and life cycle(chain of handlers, middlewares), you can optionally register it as a middleware and use the package-level `sessions.Get` to retrieve the stored-to-context session.

The `Sessions` struct value contains the `Handler` method which can be used to return an `iris.Handler` to be registered as middleware.

```
import "github.com/kataras/iris/v12/sessions"

sess := sessions.New(sessions.Config{...})

app := iris.New()
app.Use(sess.Handler())

app.Get("/path", func(ctx iris.Context){
    session := sessions.Get(ctx)
    // [use session...]
})
```

Also, if the sessions manager's `Config.AllowReclaim` is `true` then you can still call `sess.Start` as many times as you want in the same request life cycle without the need of registering it as a middleware.

Navigate to https://github.com/kataras/iris/tree/master/_examples/sessions for more examples about the sessions subpackage.

Continue by reading the [Flash Messages](#) section.

Sessions Database

Sometimes you need a backend storage, i.e redis, which will keep your session data on server restarts and scale horizontally.

Registering a session database can be done through a single call of `sessions.UseDatabase(database)` .

Iris implements three (3) builtin session databases for [redis](#), [badger](#) and [boltdb](#). These session databases are implemented via the [iris/sessions/sessiondb](#) subpackage which you'll have to import.

1. import the `github.com/kataras/iris/sessions/sessiondb/redis` or `boltdb` or `badger` ,
2. initialize the database and
3. register it

For example, to register the redis session database:

```
import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/sessions"

    // 1. Import the session database.
    "github.com/kataras/iris/v12/sessions/sessiondb/redis"
)

// 2. Initialize the database.
// These are the default values,
// you can replace them based on your running redis' server settings:
db := redis.New(redis.Config{
    Network:    "tcp",
    Addr:       "127.0.0.1:6379",
    Timeout:    time.Duration(30) * time.Second,
    MaxActive:  10,
    Password:   "",
    Database:   "",
    Prefix:     "",
    Delim:      "- ",
    Driver:     redis.Redigo(), // redis.Radix() can be used instead.
})

// Optionally configure the underline driver:
// driver := redis.Redigo()
// driver.MaxIdle = ...
// driver.IdleTimeout = ...
// driver.Wait = ...
// redis.Config {Driver: driver}

// Close connection when control+C/cmd+C
iris.RegisterOnInterrupt(func() {
    db.Close()
})

sess := sessions.New(sessions.Config{
    Cookie:      "sessionscookieid",
    AllowReclaim: true,
})

// 3. Register it.
sess.UseDatabase(db)

// [...]
```

boltdb

```
import "os"
import "github.com/kataras/iris/v12/sessions/sessiondb/boltdb"

db, err := boltdb.New("./sessions.db", os.FileMode(0750))
```

badger

```
import "github.com/kataras/iris/v12/sessions/sessiondb/badger"
```

```
db, err := badger.New("./data")
```


Flash Messages

Sometimes you need to temporarily store data between requests of the same user, such as error or success messages after a form submission. The Iris sessions package supports flash messages.

As we've seen the [Sessions](#) chapter, you initialize a session like this:

```
import "github.com/kataras/iris/v12/sessions"

sess := sessions.New(sessions.Config{Cookie: "cookieName", ...})
```

```
// [app := iris.New...]

app.Get("/path", func(ctx iris.Context) {
    session := sess.Start(ctx)
    // [...]
})
```

The `Session` contains the following methods to store, retrieve and remove flash messages.

```
SetFlash(key string, value interface{})
HasFlash() bool
GetFlashes() map[string]interface{}

PeekFlash(key string) interface{}
GetFlash(key string) interface{}
GetFlashString(key string) string
GetFlashStringDefault(key string, defaultValue string) string

DeleteFlash(key string)
ClearFlashes()
```

The method names are self-explained. For example, if you need to get a message and remove it at the next request use the `GetFlash` . When you only need to retrieve but don't remove then use the `PeekFlash` .

Flash messages are not stored in a database.

Websockets

WebSocket is a protocol that enables two-way persistent communication channels over TCP connections. It is used for applications such as chat, stock tickers, games, anywhere you want real-time functionality in a web application.

Use WebSockets when you need to work directly with a socket connection. For example, you might need the best possible performance for a real-time game.

First of all, read the [kataras/neffos wiki](#) to master the new websocket library built for net/http and Iris.

It comes pre-installed with Iris, however you can install it separately by executing the following shell command.

```
$ go get github.com/kataras/neffos@v0.0.8
```

Continue by reading how you can register the neffos websocket server into your Iris application.

A list of comprehensive examples working with websockets can be found at: https://github.com/kataras/iris/tree/master/_examples/websocket.

The [iris/websocket](#) subpackage contains (only) Iris-specific migrations and helpers for the [neffos websocket framework](#) one.

For example, to gain access of the request's `Context` you can call the `websocket.GetContext(Conn)` from inside an event message handler/callback:

```
// GetContext returns the Iris Context from a websocket connection.
func GetContext(c *neffos.Conn) Context
```

To register a websocket `neffos.Server` to a route use the `websocket.Handler` function:

```
// IDGenerator is an iris-specific IDGenerator for new connections.
type IDGenerator func(Context) string

// Handler returns an Iris handler to be served in a route of an Iris application.
// Accepts the neffos websocket server as its first input argument
// and optionally an Iris-specific `IDGenerator` as its second one.
func Handler(s *neffos.Server, IDGenerator ...IDGenerator) Handler
```

Usage

```
import (
    "github.com/kataras/neffos"
    "github.com/kataras/iris/v12/websocket"
)

// [...]

onChat := func(ns *neffos.NSConn, msg neffos.Message) error {
    ctx := websocket.GetContext(ns.Conn)
    // [...]
    return nil
}

app := iris.New()
ws := neffos.New(websocket.DefaultGorillaUpgrader, neffos.Namespaces{
    "default": neffos.Events {
        "chat": onChat,
    },
})
app.Get("/websocket_endpoint", websocket.Handler(ws))
```

Websocket Controller

Iris has a trivial way of registering websocket events via a Go structure. The websocket controller is part of the [MVC](#) features.

Iris has its own `iris/mvc/Application.HandleWebsocket(v interface{}) *neffos.Struct` to register controllers in existing Iris MVC applications(offering a fully featured dependency injection container for request values and static services).

```
// HandleWebsocket handles a websocket specific controller.
// Its exported methods are the events.
// If a "Namespace" field or method exists then namespace is set,
// otherwise empty namespace will be used for this controller.
//
// Note that a websocket controller is registered and ran under
// a connection connected to a namespace
// and it cannot send HTTP responses on that state.
// However all static and dynamic dependencies behave as expected.
func (*mvc.Application) HandleWebsocket(controller interface{}) *neffos.Struct
```

Let's see a usage example, we want to bind the `OnNamespaceConnected` , `OnNamespaceDisconnect` built-in events and a custom `"OnChat"` event with our controller's methods.

1. We create the controller by declaring a NSConn type field as `stateless` and write the methods we need.

```
type websocketController struct {
    *neffos.NSConn `stateless:"true"`
    Namespace string

    Logger MyLoggerInterface
}

func (c *websocketController) OnNamespaceConnected(msg neffos.Message) error {
    return nil
}

func (c *websocketController) OnNamespaceDisconnect(msg neffos.Message) error {
    return nil
}

func (c *websocketController) OnChat(msg neffos.Message) error {
    return nil
}
```

Iris is smart enough to catch the `Namespace string` struct field to use it to register the controller's methods as events for that namespace, alternatively you can create a controller method of `Namespace() string { return "default" }` or use the `HandleWebsocket` 's return value to `.SetNamespace("default")` , it's up to you.

2. We initialize our MVC application targets to a websocket endpoint, as we used to do with regular HTTP Controllers for HTTP routes.

```
import (
    // [...]
    "github.com/kataras/iris/v12/mvc"
)
// [app := iris.New...]

mvcApp := mvc.New(app.Party("/websocket_endpoint"))
```

3. We register our dependencies, if any.

```
mvcApp.Register(
    &prefixedLogger{prefix: "DEV"},
)
```

4. We register one or more websocket controllers, each websocket controller maps to one namespace (just one is enough, as in most of the cases you don't need more, but that depends on your app's needs and requirements).

```
mvcApp.HandleWebsocket(&websocketController{Namespace: "default"})
```

5. Next, we continue by mapping the mvc application as a connection handler to a websocket server (you may use more than one mvc applications per websocket server via `neffos.JoinConnHandlers(mvcApp1, mvcApp2)`).

```
websocketServer := neffos.New(websocket.DefaultGorillaUpgrader, mvcApp)
```

6. And the last step is to register that server to our endpoint through a normal `.Get` method.

```
mvcApp.Router.Get("/", websocket.Handler(websocketServer))
```

Dependency Injection

The subpackage `hero` contains features for binding any object or function that handlers can accept on their input arguments, these are called dependencies.

A dependency can be either `Static` for things like Services or `Dynamic` for values that depend on the incoming request.

With Iris you get truly safe bindings. It is blazing-fast, near to raw handlers performance because Iris tries to calculates everything before the server even goes online!

Iris provides built-in dependencies to match route's parameters with func input arguments that you can use right away.

To use this feature you should import the hero subpackage:

```
import (
    // [...]
    "github.com/kataras/iris/v12/hero"
)
```

And use its `hero.Handler` package-level function to build a handler from a function that can accept dependencies and send response from its output, like this:

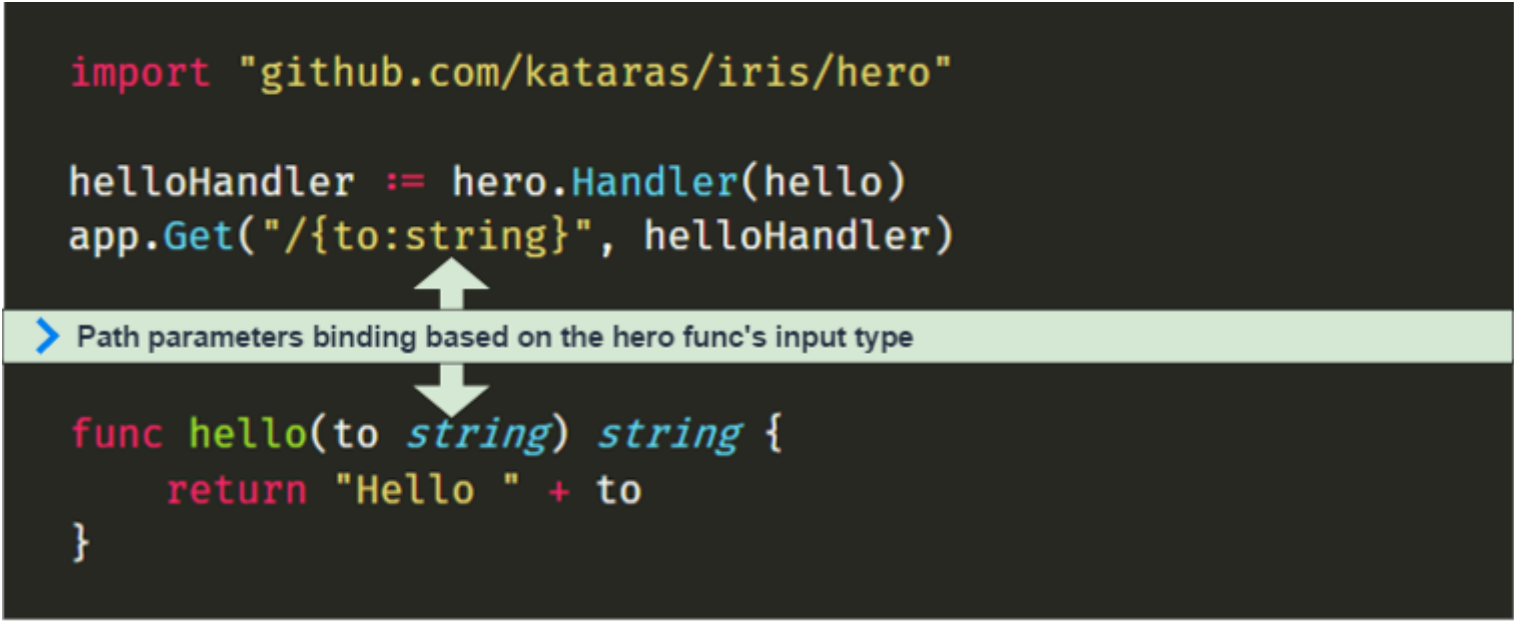
```
func printFromTo(from, to string) string { /* [...]*/ }

// [...]
app.Get("/{from}/{to}", hero.Handler(printFromTo))
```

As you've seen above the `iris.Context` input argument is totally optional. Of course you can still declare it as **first input argument** - Iris is smart enough to bind it as well without any hassle.

Below you will see some screenshots designed to facilitate your understanding:

1. Path Parameters - Built-in Dependencies



2. Services - Static Dependencies

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

type Service interface {
    SayHello(to string) string
}

type myTestService struct {
    prefix string
}

func (s *myTestService) SayHello(to string) string {
    return s.prefix + " " + to
}

hero.Register(&myTestService{
    prefix: "Service: Hello",
})

helloServiceHandler := hero.Handler(helloService)
app.Get("/{to:string}", helloServiceHandler)
```

> Service binding using the Register method.
&myTestService completes the Service, therefore hero func can accept a Service

```
func helloService(to string, service Service) string {
    return service.SayHello(to)
}
```

3. Per-Request - Dynamic Dependencies

```
type LoginForm struct {
    Username string `form:"username"`
    Password string `form:"password"`
}

hero.Register(func(ctx iris.Context) (form LoginForm) {
    // it binds the "form" with the
    // x-www-form-urlencoded form data,
    // sent by the client, and returns it.
    ctx.ReadForm(&form)
    return
})

loginHandler := hero.Handler(login)
app.Post("/login", loginHandler)
```

> Form binding using the Register method.
The registered value is a function with one input 'iris.Context' and one single out value. That output value is a type of 'LoginForm', therefore the consumer, the hero login func, can accept a 'LoginForm'

```
func login(form LoginForm) string {
    return "Hello " + form.Username
}
```

> When a `func(iris.Context) <TValue>` is passed on the Register method, we call it a "dynamic binding", a "dynamic dependency" that depends on the current request's Context

In addition the hero subpackage adds support to send responses through the **output values** of a function, for example:

- if the return value is `string` then it will send that string as the response's body.
- If it's an `int` then it will send it as a status code.
- If it's an `error` then it will set a bad request with that error as its reason.
- If it's an `error` and an `int` then the error code is the output integer instead of 400(bad request).
- If it's a custom `struct` then it sent as a JSON, when a Content-Type header is not already set.
- If it's a custom `struct` and a `string` then the second output value, string, it will be the Content-Type and so on.

```
func myHandler(...dependencies) string |
    (string, string) |
    (string, int) |
    int |
    (int, string) |
    (string, error) |
    error |
    (int, error) |
    (any, bool) |
    (customStruct, error) |
    customStruct |
    (customStruct, int) |
    (customStruct, string) |
    hero.Result |
    (hero.Result, error) {

    return a_response
}
```

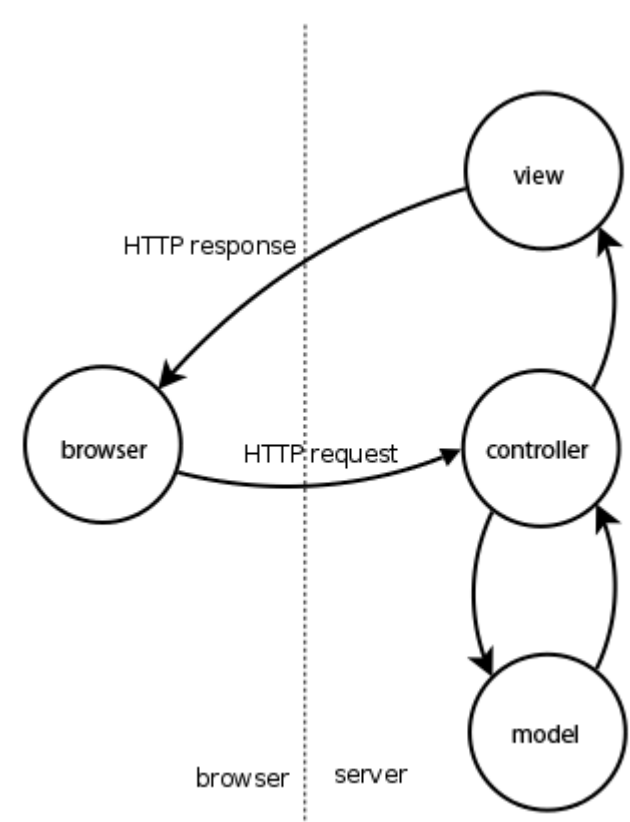
The `hero.Result` is an interface which help custom structs to be rendered using custom logic through their `Dispatch(ctx iris.Context)` .

```
type Result interface {
    Dispatch(ctx iris.Context)
}
```

Honestly, `hero funcs` are very easy to understand and when you start using them **you never go back**.

Later on you'll see how this knowledge will help you to craft an application using the MVC architectural pattern which Iris provides wonderful API for it.

MVC



Using Iris MVC for code reuse.

By creating components that are independent of one another, developers are able to reuse components quickly and easily in other applications. The same (or similar) view for one application can be refactored for another application with different data because the view is simply handling how the data is being displayed to the user.

Iris has **first-class support for the MVC (Model View Controller) architectural pattern**, you'll not find these stuff anywhere else in the Go world. You will have to import the `iris/mvc` subpackage.

```
import "github.com/kataras/iris/v12/mvc"
```

Iris web framework supports Request data, Models, Persistence Data and Binding with the fastest possible execution.

If you're new to back-end web development read about the MVC architectural pattern first, a good start is that [wikipedia article](#).

Characteristics

All HTTP Methods are supported, for example if want to serve `GET` then the controller should have a function named `Get()` , you can define more than one method function to serve in the same Controller.

Serve custom controller's struct's methods as handlers with custom paths(even with regex parametermized path) via the `BeforeActivation` custom event callback, per-controller. Example:


```

import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/mvc"
)

func main() {
    app := iris.New()
    mvc.Configure(app.Party("/root"), myMVC)
    app.Run(iris.Addr(":8080"))
}

func myMVC(app *mvc.Application) {
    // app.Register(...)
    // app.Router.Use/UseGlobal/Done(...)
    app.Handle(new(MyController))
}

type MyController struct {}

func (m *MyController) BeforeActivation(b mvc.BeforeActivation) {
    // b.Dependencies().Add/Remove
    // b.Router().Use/UseGlobal/Done
    // and any standard Router API call you already know

    // 1-> Method
    // 2-> Path
    // 3-> The controller's function name to be parsed as handler
    // 4-> Any handlers that should run before the MyCustomHandler
    b.Handle("GET", "/something/{id:long}", "MyCustomHandler", anyMiddleware...)
}

// GET: http://localhost:8080/root
func (m *MyController) Get() string {
    return "Hey"
}

// GET: http://localhost:8080/root/something/{id:long}
func (m *MyController) MyCustomHandler(id int64) string {
    return "MyCustomHandler says Hey"
}

```

Persistence data inside your Controller struct (share data between requests) by defining services to the Dependencies or have a `Singleton` controller scope.

Share the dependencies between controllers or register them on a parent MVC Application, and ability to modify dependencies per-controller on the `BeforeActivation` optional event callback inside a Controller, i.e `func(c *MyController) BeforeActivation(b mvc.BeforeActivation) { b.Dependencies().Add/Remove(...) }` .

Access to the `Context` as a controller's field(no manual binding is needed) i.e `Ctx iris.Context` or via a method's input argument, i.e `func(ctx iris.Context, otherArguments...) .`

Models inside your Controller struct (set-ed at the Method function and rendered by the View). You can return models from a controller's method or set a field in the request lifecycle and return that field to another method, in the same request lifecycle.

Flow as you used to, mvc application has its own `Router` which is a type of `iris/router.Party` , the standard iris api. `Controllers` can be registered to any `Party` , including Subdomains, the Party's begin and done handlers work as expected.

Optional `BeginRequest(ctx)` function to perform any initialization before the method execution, useful to call middlewares or when many methods use the same collection of data.

Optional `EndRequest(ctx)` function to perform any finalization after any method executed.

Inheritance, recursively, e.g. our [mvc session-controller example](#), has the `Session *sessions.Session` as struct field which is filled by the sessions manager's `Start` as a dynamic dependency to the MVC Application: `mvcApp.Register(sessions.New(sessions.Config{Cookie: "iris_session_id"}).Start)` .

Access to the dynamic path parameters via the controller's methods' input arguments, no binding is needed. When you use the Iris' default syntax to parse handlers from a controller, you need to suffix the methods with the `By` word, uppercase is a new sub path. Example:

If `mvc.New(app.Party("/user")).Handle(new(user.Controller))`

- `func(*Controller) Get()` - `GET:/user` .
- `func(*Controller) Post()` - `POST:/user` .
- `func(*Controller) GetLogin()` - `GET:/user/login`
- `func(*Controller) PostLogin()` - `POST:/user/login`
- `func(*Controller) GetProfileFollowers()` - `GET:/user/profile/followers`

- `func(*Controller) PostProfileFollowers()` - `POST:/user/profile/followers`
- `func(*Controller) GetBy(id int64)` - `GET:/user/{param:long}`
- `func(*Controller) PostBy(id int64)` - `POST:/user/{param:long}`

```
if mvc.New(app.Party("/profile")).Handle(new(profile.Controller))
```

- `func(*Controller) GetBy(username string)` - `GET:/profile/{param:string}`

```
if mvc.New(app.Party("/assets")).Handle(new(file.Controller))
```

- `func(*Controller) GetByWildard(path string)` - `GET:/assets/{param:path}`

Supported types for method functions receivers: int, int64, bool and string.

Optionally, response via output arguments, like we've shown at the [Dependency Injection](#) chapter. E.g.

```
func(c *ExampleController) Get() string |
    (string, string) |
    (string, int) |
    int |
    (int, string) |
    (string, error) |
    error |
    (int, error) |
    (any, bool) |
    (customStruct, error) |
    customStruct |
    (customStruct, int) |
    (customStruct, string) |
    mvc.Result or (mvc.Result, error)
```

Where `mvc.Result` is a `hero.Result` type alias, which is this interface one:

```
type Result interface {
    // Dispatch should sends the response to the context's response writer.
    Dispatch(ctx iris.Context)
}
```

Example

This example is equivalent to the https://github.com/kataras/iris/blob/master/_examples/hello-world/main.go

It seems that additional code you have to write doesn't worth it but remember that, this example does not make use of iris mvc features like the Model, Persistence or the View engine neither the Session, it's very simple for learning purposes, probably you'll never use such as simple controller anywhere in your app.

The cost we have on this example for using MVC on the `/hello` path which serves JSON is ~2MB per 20MB throughput on my personal laptop, it's tolerated for the majority of the applications but you can choose what suits you best with Iris, low-level handlers: performance or high-level controllers: easier to maintain and smaller codebase on large applications.

Read the comments very careful

```

package main

import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/mvc"

    "github.com/kataras/iris/v12/middleware/logger"
    "github.com/kataras/iris/v12/middleware/recover"
)

func main() {
    app := iris.New()
    // Optionally, add two built'n handlers
    // that can recover from any http-relative panics
    // and log the requests to the terminal.
    app.Use(recover.New())
    app.Use(logger.New())

    // Serve a controller based on the root Router, "/".
    mvc.New(app).Handle(new(ExampleController))

    // http://localhost:8080
    // http://localhost:8080/ping
    // http://localhost:8080/hello
    // http://localhost:8080/custom_path
    app.Run(iris.Addr(":8080"))
}

// ExampleController serves the "/", "/ping" and "/hello".
type ExampleController struct{}

// Get serves
// Method: GET
// Resource: http://localhost:8080
func (c *ExampleController) Get() mvc.Result {
    return mvc.Response{
        ContentType: "text/html",
        Text:        "<h1>Welcome</h1>",
    }
}

// GetPing serves
// Method: GET
// Resource: http://localhost:8080/ping
func (c *ExampleController) GetPing() string {
    return "pong"
}

// GetHello serves
// Method: GET
// Resource: http://localhost:8080/hello
func (c *ExampleController) GetHello() interface{} {
    return map[string]string{"message": "Hello Iris!"}
}

// BeforeActivation called once, before the controller adapted to the main application
// and of course before the server ran.
// After version 9 you can also add custom routes for a specific controller's methods.
// Here you can register custom method's handlers
// use the standard router with `ca.Router` to
// do something that you can do without mvc as well,
// and add dependencies that will be binded to
// a controller's fields or method function's input arguments.
func (c *ExampleController) BeforeActivation(b mvc.BeforeActivation) {
    anyMiddlewareHere := func(ctx iris.Context) {

```

```

        ctx.Application().Logger().Warnf("Inside /custom_path")
        ctx.Next()
    }

    b.Handle(
        "GET",
        "/custom_path",
        "CustomHandlerWithoutFollowingTheNamingGuide",
        anyMiddlewareHere,
    )

    // or even add a global middleware based on this controller's router,
    // which in this example is the root "/":
    // b.Router().Use(myMiddleware)
}

// CustomHandlerWithoutFollowingTheNamingGuide serves
// Method:    GET
// Resource: http://localhost:8080/custom_path
func (c *ExampleController) CustomHandlerWithoutFollowingTheNamingGuide() string {
    return "hello from the custom handler without following the naming guide"
}

// GetUserBy serves
// Method:    GET
// Resource: http://localhost:8080/user/{username:string}
// By is a reserved "keyword" to tell the framework that you're going to
// bind path parameters in the function's input arguments, and it also
// helps to have "Get" and "GetBy" in the same controller.
//
// func (c *ExampleController) GetUserBy(username string) mvc.Result {
//     return mvc.View{
//         Name: "user/username.html",
//         Data: username,
//     }
// }

/* Can use more than one, the factory will make sure
that the correct http methods are being registered for each route
for this controller, uncomment these if you want:

func (c *ExampleController) Post() {}
func (c *ExampleController) Put() {}
func (c *ExampleController) Delete() {}
func (c *ExampleController) Connect() {}
func (c *ExampleController) Head() {}
func (c *ExampleController) Patch() {}
func (c *ExampleController) Options() {}
func (c *ExampleController) Trace() {}
*/

/*
func (c *ExampleController) All() {}
//      OR
func (c *ExampleController) Any() {}

func (c *ExampleController) BeforeActivation(b mvc.BeforeActivation) {
    // 1 -> the HTTP Method
    // 2 -> the route's path
    // 3 -> this controller's method name that should be handler for that route.
    b.Handle("GET", "/mypath/{param}", "DoIt", optionalMiddlewareHere...)
}

// After activation, all dependencies are set-ed - so read only access on them

```

```
// but still possible to add custom controller or simple standard handlers.
func (c *ExampleController) AfterActivation(a mvc.AfterActivation) {}

*/
```

Every `exported` func prefixed with an HTTP Method(`Get` , `Post` , `Put` , `Delete` ...) in a controller is callable as an HTTP endpoint. In the sample above, all funcs writes a string to the response. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `http://localhost:8080/helloworld` , and combines the protocol used: HTTP, the network location of the web server (including the TCP port): `localhost:8080` and the target URI `/helloworld` .

The first comment states this is an [HTTP GET](#) method that is invoked by appending "/helloworld" to the base URL. The third comment specifies an [HTTP GET](#) method that is invoked by appending "/helloworld/welcome" to the URL.

Controller knows how to handle the "name" on `GetBy` or the "name" and "numTimes" at `GetWelcomeBy` , because of the `By` keyword, and builds the dynamic route without boilerplate; the third comment specifies an [HTTP GET](#) dynamic method that is invoked by any URL that starts with "/helloworld/welcome" and followed by two more path parts, the first one can accept any value and the second can accept only numbers, i,e: "[http://localhost:8080/helloworld/welcome/golang/32719](#)", otherwise a [404 Not Found HTTP Error](#) will be sent to the client instead.

The [_examples/mvc](#) and [mvc/controller_test.go](#) files explain each feature with simple paradigms, they show how you can take advantage of the Iris MVC Binder, Iris MVC Models and many more...

For websocket controller go ahead to the [Websockets](#) chapter instead.

Testing

Iris offers an incredible support for the [httpexpect](#), a Testing Framework for web applications. The `iris/httpptest` subpackage provides helpers for Iris + httpexpect.

if you prefer the Go's standard [net/http/httptest](#) package, you can still use it. Iris as much every http web framework is compatible with any external tool for testing, at the end it's HTTP.

Basic Authentication

In our first example we will use the `iris/httpptest` to test Basic Authentication.

1. The `main.go` source file looks like that:

```
package main

import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/middleware/basicauth"
)

func newApp() *iris.Application {
    app := iris.New()

    authConfig := basicauth.Config{
        Users: map[string]string{"myusername": "mypassword"},
    }

    authentication := basicauth.New(authConfig)

    app.Get("/", func(ctx iris.Context) { ctx.Redirect("/admin") })

    needAuth := app.Party("/admin", authentication)
    {
        //http://localhost:8080/admin
        needAuth.Get("/", h)
        // http://localhost:8080/admin/profile
        needAuth.Get("/profile", h)

        // http://localhost:8080/admin/settings
        needAuth.Get("/settings", h)
    }

    return app
}

func h(ctx iris.Context) {
    username, password, _ := ctx.Request().BasicAuth()
    // third parameter ^ will be always true because the middleware
    // makes sure for that, otherwise this handler will not be executed.

    ctx.Writef("%s %s:%s", ctx.Path(), username, password)
}

func main() {
    app := newApp()
    app.Run(iris.Addr(":8080"))
}
```

2. Now, create a `main_test.go` file and copy-paste the following.

```
package main

import (
    "testing"

    "github.com/kataras/iris/v12/httpptest"
)

func TestNewApp(t *testing.T) {
    app := newApp()
    e := httpptest.New(t, app)

    // redirects to /admin without basic auth
    e.GET("/").Expect().Status(httpptest.StatusUnauthorized)
    // without basic auth
    e.GET("/admin").Expect().Status(httpptest.StatusUnauthorized)

    // with valid basic auth
    e.GET("/admin").WithBasicAuth("myusername", "mypassword").Expect().
        Status(httpptest.StatusOK).Body().Equal("/admin myusername:mypassword")
    e.GET("/admin/profile").WithBasicAuth("myusername", "mypassword").Expect().
        Status(httpptest.StatusOK).Body().Equal("/admin/profile myusername:mypassword")
    e.GET("/admin/settings").WithBasicAuth("myusername", "mypassword").Expect().
        Status(httpptest.StatusOK).Body().Equal("/admin/settings myusername:mypassword")

    // with invalid basic auth
    e.GET("/admin/settings").WithBasicAuth("invalidusername", "invalidpassword").
        Expect().Status(httpptest.StatusUnauthorized)
}
```

3. Open your command line and execute:

```
$ go test -v
```

Other example: cookies


```
package main

import (
    "fmt"
    "testing"

    "github.com/kataras/iris/v12/httpptest"
)

func TestCookiesBasic(t *testing.T) {
    app := newApp()
    e := httpptest.New(t, app, httpptest.URL("http://example.com"))

    cookieName, cookieValue := "my_cookie_name", "my_cookie_value"

    // Test Set A Cookie.
    t1 := e.GET(fmt.Sprintf("/cookies/%s/%s", cookieName, cookieValue)).
        Expect().Status(httpptest.StatusOK)
    // Validate cookie's existence, it should be available now.
    t1.Cookie(cookieName).Value().Equal(cookieValue)
    t1.Body().Contains(cookieValue)

    path := fmt.Sprintf("/cookies/%s", cookieName)

    // Test Retrieve A Cookie.
    t2 := e.GET(path).Expect().Status(httpptest.StatusOK)
    t2.Body().Equal(cookieValue)

    // Test Remove A Cookie.
    t3 := e.DELETE(path).Expect().Status(httpptest.StatusOK)
    t3.Body().Contains(cookieName)

    t4 := e.GET(path).Expect().Status(httpptest.StatusOK)
    t4.Cookies().Empty()
    t4.Body().Empty()
}
```

```
$ go test -v -run=TestCookiesBasic$
```

Iris web framework itself uses this package to test itself. In the [_examples repository directory](#) you will find some useful tests as well. For more information please take a look and read the [httpexpect's documentation](#).

Starter Kits

Project	Description	Stars	Author
peterq/pan-light	Baidu network disk unlimited speed client, golang + qt5, cross-platform graphical interface	9083	@peterq
mlogclub/bbs-go	Golang-based community system	349	@mlogclub
snowlyg/IrisApiProject	Iris + gorm + JWT + sqllite3	282	@snowlyg
mohuishou/scuplus-go	WeChat applet Backend API	58	@mohuishou
menghx/levante	BLOG powered by Iris!	6	@menghx
zuoyanart/pizzaCmsApi	RESTful power by Iris	37	@zuoyanart
wx85278161/go-iris-vue	Iris + Vue + Casbin + JWT	46	@wx85278161
yz124/superstar	Iris + xorm to implement the star library	91	@yz124
pusher.com	A realtime API monitor written with go	5	@neoighodaro
jebzmos4/Iris-golang	A basic CRUD API in golang with Iris	8	@jebzmos4
gauravtiwari/go_iris_app	Basic web app built in Iris	21	@gauravtiwari
iris-contrib/Iris-Mini-Social-Network	A mini social-network created with the awesome Iris💕💕	47	@iris-contrib
iris-contrib/iris-starter-kit	Iris isomorphic react/hot reloadable/redux/css-modules	46	@iris-contrib
TimothyYe/iris-demo	Iris demo project	2	@TimothyYe
ionutvilie/react-ts	Demo project with react using typescript and Iris	9	@ionutvilie
iris-contrib/parrot	Self-hosted Localization Management Platform built with Iris and Angular	9	@iris-contrib
iris-contrib/cloud-native-go	Iris + Docker and Kubernetes	27	@iris-contrib
nanobox.io	Quickstart for Iris with Nanobox	9	@nanobox-io
hasura.io	A Hasura starter project with a ready to deploy Golang hello-world web app with Iris	9	@k8s-platform-hub

Prepare a cup of coffee or tea, whatever pleases you the most, and read some articles we found for you.

- [Golang Iris Web Course 2019](#)
- [How to write a Go API Part 1: A Webserver With Iris](#)
- [How to write a Go API Part 2: Database Integration With GORM](#)
- [How to write a Go API Part 3: Testing With Dockertest](#)
- [A URL Shortener Service using Go, Iris and Bolt \(Updated\)](#)
- [CRUD REST API in Iris \(a framework for golang\)](#)
- [Monitor APIs in realtime using Go and Iris](#)
- [A Todo MVC Application using Iris and Vue.js](#)
- [A Hasura starter project with a ready to deploy Golang hello-world web app with IRIS](#)
- [Top 6 web frameworks for Go as of 2017](#)
- [Iris Go Framework + MongoDB](#)
- [How to build a file upload form using DropzoneJS and Go](#)
- [How to display existing files on server using DropzoneJS and Go](#)
- [Iris, a modular web framework](#)
- [Go vs .NET Core in terms of HTTP performance](#)
- [Iris Go vs .NET Core Kestrel in terms of HTTP performance](#)
- [How to Turn an Android Device into a Web Server](#)

Video Courses

Description	Link	Author	Year
Web Course	https://bit.ly/web-course-2019	TechMaster	2019
Quick Start with Iris	https://bit.ly/2wQlrJw	J-Secur1ty	2019
Installing Iris	https://bit.ly/2KhgB1J	WarnabiruTV	2018
Iris & Mongo DB Complete	https://bit.ly/2lcXZOu	Musobar Media	2018
Getting Started with Iris	https://bit.ly/2XGafMv	stephgdesign	2018

Contact via [e-mail](#) to help you out.

- Do you want to write an Article about Iris but you needed a kind of assistance or help?
- Have you already written one that's not listed here?
- Did you create a video course about Iris that it's not listed here?