Gin Web Framework

Run Tests passing codecov 98% go report A+

GO reference gitter join chat used by 4.1k projects

code helpers 64 release v1.8.1 TODOs 6

Gin is a web framework written in Go (Golang). It features a martini-like API with performance that is up to 40 times faster thanks to httprouter. If you need performance and good productivity, you will love Gin.

Contents

Installation

To install Gin package, you need to install Go and set your Go workspace first.

1. The first need Go installed (**version 1.14+ is required**), then you can use the below Go command to install Gin.

```
$ go get -u github.com/gin-gonic/gin
```

2. Import it in your code:

```
import "github.com/gin-gonic/gin"
```

3. (Optional) Import `net/http`. This is required for example if using constants such as `http.StatusOK`.

```
import "net/http"
```

Quick start

```
# assume the following codes in example.go file
$ cat example.go
```

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })
    r.Run() // listen and serve on 0.0.0.0:8080 (for windows "localhost:8080")
}
```

```
# run example.go and visit 0.0.0.0:8080/ping (for windows "localhost:8080/ping") on
browser
$ go run example.go
```

Benchmarks

Gin uses a custom version of HttpRouter

See all benchmarks

Benchmark name	(1)	(2)	(3)	(4)
BenchmarkGin_GithubAll	**43550**	**27364 ns/op**	**0 B/op**	**0 allocs/op**
BenchmarkAce_GithubAll	40543	29670 ns/op	0 B/op	0 allocs/op
BenchmarkAero_GithubAll	57632	20648 ns/op	0 B/op	0 allocs/op
BenchmarkBear_GithubAll	9234	216179 ns/op	86448 B/op	943 allocs/op
BenchmarkBeego_GithubAll	7407	243496 ns/op	71456 B/op	609 allocs/op
BenchmarkBone_GithubAll	420	2922835 ns/op	720160 B/op	8620 allocs/op
BenchmarkChi_GithubAll	7620	238331 ns/op	87696 B/op	609 allocs/op
BenchmarkDenco_GithubAll	18355	64494 ns/op	20224 B/op	167 allocs/op
BenchmarkEcho_GithubAll	31251	38479 ns/op	0 B/op	0 allocs/op
BenchmarkGocraftWeb_GithubAll	4117	300062 ns/op	131656 B/op	1686 allocs/op

BenchmarkGoji_GithubAll	3274	416158 ns/op	56112 B/op	334 allocs/op
BenchmarkGojiv2_GithubAll	1402	870518 ns/op	352720 B/op	4321 allocs/op
BenchmarkGoJsonRest_GithubAll	2976	401507 ns/op	134371 B/op	2737 allocs/op
BenchmarkGoRestful_GithubAll	410	2913158 ns/op	910144 B/op	2938 allocs/op
BenchmarkGorillaMux_GithubAll	346	3384987 ns/op	251650 B/op	1994 allocs/op
BenchmarkGowwwRouter_GithubAll	10000	143025 ns/op	72144 B/op	501 allocs/op
BenchmarkHttpRouter_GithubAll	55938	21360 ns/op	0 B/op	0 allocs/op
BenchmarkHttpTreeMux_GithubAll	10000	153944 ns/op	65856 B/op	671 allocs/op
BenchmarkKocha_GithubAll	10000	106315 ns/op	23304 B/op	843 allocs/op
BenchmarkLARS_GithubAll	47779	25084 ns/op	0 B/op	0 allocs/op
BenchmarkMacaron_GithubAll	3266	371907 ns/op	149409 B/op	1624 allocs/op
BenchmarkMartini_GithubAll	331	3444706 ns/op	226551 B/op	2325 allocs/op
BenchmarkPat_GithubAll	273	4381818 ns/op	1483152 B/op	26963 allocs/op
BenchmarkPossum_GithubAll	10000	164367 ns/op	84448 B/op	609 allocs/op
BenchmarkR2router_GithubAll	10000	160220 ns/op	77328 B/op	979 allocs/op
BenchmarkRivet_GithubAll	14625	82453 ns/op	16272 B/op	167 allocs/op
BenchmarkTango_GithubAll	6255	279611 ns/op	63826 B/op	1618 allocs/op
BenchmarkTigerTonic_GithubAll	2008	687874 ns/op	193856 B/op	4474 allocs/op
BenchmarkTraffic_GithubAll	355	3478508 ns/op	820744 B/op	14114 allocs/op
BenchmarkVulcan_GithubAll	6885	193333 ns/op	19894 B/op	609 allocs/op

- (1): Total Repetitions achieved in constant time, higher means more confident result
- (2): Single Repetition Duration (ns/op), lower is better
- (3): Heap Memory (B/op), lower is better
- (4): Average Allocations per Repetition (allocs/op), lower is better

Gin v1. stable

- ✅ Zero allocation router.
- ✅ Still the fastest http router and framework. From routing to writing.
- ✅ Complete suite of unit tests.
- ✅ Battle tested.
- ✅ API frozen, new releases will not break your code.

Build with json replacement

Gin uses `encoding/json` as default json package but you can change it by build from other tags.

jsoniter

```
$ go build -tags=jsoniter .
```

go-json

```
$ go build -tags=go_json .
```

Build without `MsgPack` rendering feature

Gin enables `MsgPack` rendering feature by default. But you can disable this feature by specifying `nomsgpack` build tag.

```
$ go build -tags=nomsgpack .
```

This is useful to reduce the binary size of executable files. See the detail information.

API Examples

You can find a number of ready-to-run examples at Gin examples repository.

Using GET, POST, PUT, PATCH, DELETE and OPTIONS

```
func main() {
    // Creates a gin router with default middleware:
    // logger and recovery (crash-free) middleware
    router := gin.Default()

    router.GET("/someGet", getting)
    router.POST("/somePost", posting)
    router.PUT("/somePut", putting)
    router.DELETE("/someDelete", deleting)
    router.PATCH("/somePatch", patching)
    router.HEAD("/someHead", head)
    router.OPTIONS("/someOptions", options)

    // By default it serves on :8080 unless a
    // PORT environment variable was defined.
    router.Run()
    // router.Run(":3000") for a hard coded port
}
```

Parameters in path

```
func main() {
    router := gin.Default()
    // This handler will match /user/john but will not match /user/ or /user
    router.GET("/user/:name", func(c *gin.Context) {
        name := c.Param("name")
        c.String(http.StatusOK, "Hello %s", name)
    })

    // However, this one will match /user/john/ and also /user/john/send
    // If no other routers match /user/john, it will redirect to /user/john/
    router.GET("/user/:name/*action", func(c *gin.Context) {
        name := c.Param("name")
        action := c.Param("action")
        message := name + " is " + action
        c.String(http.StatusOK, message)
    })

    // For each matched request Context will hold the route definition
    router.POST("/user/:name/*action", func(c *gin.Context) {
        b := c.FullPath() == "/user/:name/*action" // true
        c.String(http.StatusOK, "%t", b)
    })

    // This handler will add a new router for /user/groups.
    // Exact routes are resolved before param routes, regardless of the order they
were defined.
    // Routes starting with /user/groups are never interpreted as /user/:name/...
routes
    router.GET("/user/groups", func(c *gin.Context) {
        c.String(http.StatusOK, "The available groups are [...]")
    })

    router.Run(":8080")
}
```

Querystring parameters

```
func main() {
    router := gin.Default()

    // Query string parameters are parsed using the existing underlying request
object.
    // The request responds to a url matching:  /welcome?firstname=Jane&lastname=Doe
    router.GET("/welcome", func(c *gin.Context) {
        firstname := c.DefaultQuery("firstname", "Guest")
        lastname := c.Query("lastname") // shortcut for
c.Request.URL.Query().Get("lastname")

        c.String(http.StatusOK, "Hello %s %s", firstname, lastname)
    })
    router.Run(":8080")
}
```

Multipart/Urlencoded Form

```
func main() {
    router := gin.Default()

    router.POST("/form_post", func(c *gin.Context) {
        message := c.PostForm("message")
        nick := c.DefaultPostForm("nick", "anonymous")

        c.JSON(200, gin.H{
            "status":  "posted",
            "message": message,
            "nick":    nick,
        })
    })
    router.Run(":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() {
    router := gin.Default()

    router.POST("/post", func(c *gin.Context) {

        id := c.Query("id")
        page := c.DefaultQuery("page", "0")
        name := c.PostForm("name")
        message := c.PostForm("message")

        fmt.Printf("id: %s; page: %s; name: %s; message: %s", id, page, name,
message)
    })
    router.Run(":8080")
}
```

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

Map as querystring or postform parameters

```
POST /post?ids[a]=1234&ids[b]=hello HTTP/1.1
Content-Type: application/x-www-form-urlencoded
```

```
names[first]=thinkerou&names[second]=tianou
```

```go
func main() {
	router := gin.Default()

	router.POST("/post", func(c *gin.Context) {

		ids := c.QueryMap("ids")
		names := c.PostFormMap("names")

		fmt.Printf("ids: %v; names: %v", ids, names)
	})
	router.Run(":8080")
}
```

```
ids: map[b:hello a:1234]; names: map[second:tianou first:thinkerou]
```

Upload files

Single file

References issue #774 and detail example code.

`file.Filename` **SHOULD NOT** be trusted. See `Content-Disposition` on MDN and #1693

> *The filename is always optional and must not be used blindly by the application: path information should be stripped, and conversion to the server file system rules should be done.*

```go
func main() {
	router := gin.Default()
	// Set a lower memory limit for multipart forms (default is 32 MiB)
	router.MaxMultipartMemory = 8 << 20  // 8 MiB
	router.POST("/upload", func(c *gin.Context) {
		// Single file
		file, _ := c.FormFile("file")
		log.Println(file.Filename)

		// Upload the file to specific dst.
		c.SaveUploadedFile(file, dst)

		c.String(http.StatusOK, fmt.Sprintf("'%s' uploaded!", file.Filename))
	})
	router.Run(":8080")
}
```

How to `curl`:

```bash
curl -X POST http://localhost:8080/upload \
  -F "file=@/Users/appleboy/test.zip" \
  -H "Content-Type: multipart/form-data"
```

Multiple files

See the detail example code.

```
func main() {
	router := gin.Default()
	// Set a lower memory limit for multipart forms (default is 32 MiB)
	router.MaxMultipartMemory = 8 << 20  // 8 MiB
	router.POST("/upload", func(c *gin.Context) {
		// Multipart form
		form, _ := c.MultipartForm()
		files := form.File["upload[]"]

		for _, file := range files {
			log.Println(file.Filename)

			// Upload the file to specific dst.
			c.SaveUploadedFile(file, dst)
		}
		c.String(http.StatusOK, fmt.Sprintf("%d files uploaded!", len(files)))
	})
	router.Run(":8080")
}
```

How to `curl`:

```
curl -X POST http://localhost:8080/upload \
  -F "upload[]=@/Users/appleboy/test1.zip" \
  -F "upload[]=@/Users/appleboy/test2.zip" \
  -H "Content-Type: multipart/form-data"
```

Grouping routes

```
func main() {
	router := gin.Default()

	// Simple group: v1
	v1 := router.Group("/v1")
	{
		v1.POST("/login", loginEndpoint)
		v1.POST("/submit", submitEndpoint)
		v1.POST("/read", readEndpoint)
	}

	// Simple group: v2
	v2 := router.Group("/v2")
	{
		v2.POST("/login", loginEndpoint)
		v2.POST("/submit", submitEndpoint)
```

```
        v2.POST("/read", readEndpoint)
    }

    router.Run(":8080")
}
```

Blank Gin without middleware by default

Use

```
r := gin.New()
```

instead of

```
// Default With the Logger and Recovery middleware already attached
r := gin.Default()
```

Using middleware

```
func main() {
    // Creates a router without any middleware by default
    r := gin.New()

    // Global middleware
    // Logger middleware will write the logs to gin.DefaultWriter even if you set
with GIN_MODE=release.
    // By default gin.DefaultWriter = os.Stdout
    r.Use(gin.Logger())

    // Recovery middleware recovers from any panics and writes a 500 if there was
one.
    r.Use(gin.Recovery())

    // Per route middleware, you can add as many as you desire.
    r.GET("/benchmark", MyBenchLogger(), benchEndpoint)

    // Authorization group
    // authorized := r.Group("/", AuthRequired())
    // exactly the same as:
    authorized := r.Group("/")
    // per group middleware! in this case we use the custom created
    // AuthRequired() middleware just in the "authorized" group.
    authorized.Use(AuthRequired())
    {
        authorized.POST("/login", loginEndpoint)
        authorized.POST("/submit", submitEndpoint)
        authorized.POST("/read", readEndpoint)

        // nested group
        testing := authorized.Group("testing")
        // visit 0.0.0.0:8080/testing/analytics
        testing.GET("/analytics", analyticsEndpoint)
    }

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

Custom Recovery behavior

```
func main() {
    // Creates a router without any middleware by default
    r := gin.New()

    // Global middleware
    // Logger middleware will write the logs to gin.DefaultWriter even if you set
with GIN_MODE=release.
    // By default gin.DefaultWriter = os.Stdout
    r.Use(gin.Logger())

    // Recovery middleware recovers from any panics and writes a 500 if there was
one.
    r.Use(gin.CustomRecovery(func(c *gin.Context, recovered interface{}) {
        if err, ok := recovered.(string); ok {
            c.String(http.StatusInternalServerError, fmt.Sprintf("error: %s", err))
        }
        c.AbortWithStatus(http.StatusInternalServerError)
    }))

    r.GET("/panic", func(c *gin.Context) {
        // panic with a string -- the custom middleware could save this to a
database or report it to the user
        panic("foo")
    })

    r.GET("/", func(c *gin.Context) {
        c.String(http.StatusOK, "ohai")
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

How to write log file

```
func main() {
    // Disable Console Color, you don't need console color when writing the logs to
file.
    gin.DisableConsoleColor()
    // Logging to a file.
    f, _ := os.Create("gin.log")
    gin.DefaultWriter = io.MultiWriter(f)

    // Use the following code if you need to write the logs to file and console at
the same time.
    // gin.DefaultWriter = io.MultiWriter(f, os.Stdout)

    router := gin.Default()
    router.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    router.Run(":8080")
}
```

Custom Log Format

```
func main() {
    router := gin.New()

    // LoggerWithFormatter middleware will write the logs to gin.DefaultWriter
    // By default gin.DefaultWriter = os.Stdout
    router.Use(gin.LoggerWithFormatter(func(param gin.LogFormatterParams) string {

        // your custom format
        return fmt.Sprintf("%s - [%s] \"%s %s %s %d %s \"%s\" %s\"\n",
                param.ClientIP,
                param.TimeStamp.Format(time.RFC1123),
                param.Method,
                param.Path,
                param.Request.Proto,
                param.StatusCode,
                param.Latency,
                param.Request.UserAgent(),
                param.ErrorMessage,
        )
    }))
    router.Use(gin.Recovery())

    router.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    router.Run(":8080")
}
```

Sample Output

```
::1 - [Fri, 07 Dec 2018 17:04:38 JST] "GET /ping HTTP/1.1 200 122.767µs "Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/71.0.3578.80 Safari/537.36" "
```

Controlling Log output coloring

By default, logs output on console should be colorized depending on the detected TTY.

Never colorize logs:

```go
func main() {
    // Disable log's color
    gin.DisableConsoleColor()

    // Creates a gin router with default middleware:
    // logger and recovery (crash-free) middleware
    router := gin.Default()

    router.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    router.Run(":8080")
}
```

Always colorize logs:

```go
func main() {
    // Force log's color
    gin.ForceConsoleColor()

    // Creates a gin router with default middleware:
    // logger and recovery (crash-free) middleware
    router := gin.Default()

    router.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    router.Run(":8080")
}
```

Model binding and validation

To bind a request body into a type, use model binding. We currently support binding of JSON, XML, YAML and standard form values (foo=bar&boo=baz).

Gin uses **go-playground/validator/v10** for validation. Check the full docs on tags usage here.

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"` .

Also, Gin provides two sets of methods for binding:

- **Type** - Must bind
  - **Methods** - `Bind` , `BindJSON` , `BindXML` , `BindQuery` , `BindYAML` , `BindHeader`
  - **Behavior** - These methods use `MustBindWith` under the hood. If there is a binding error, the request is aborted with `c.AbortWithError(400, err).SetType(ErrorTypeBind)` . This sets the response status code to 400 and the `Content-Type` header is set to `text/plain; charset=utf-8` . Note that if you try to set the response code after this, it will result in a warning `[GIN-debug] [WARNING] Headers were already written. Wanted to override status code 400 with 422` . If you wish to have greater control over the behavior, consider using the `ShouldBind` equivalent method.
- **Type** - Should bind
  - **Methods** - `ShouldBind` , `ShouldBindJSON` , `ShouldBindXML` , `ShouldBindQuery` , `ShouldBindYAML` , `ShouldBindHeader`
  - **Behavior** - These methods use `ShouldBindWith` under the hood. If there is a binding error, the error is returned and it is the developer's responsibility to handle the request and error appropriately.

When using the Bind-method, Gin tries to infer the binder depending on the Content-Type header. If you are sure what you are binding, you can use `MustBindWith` or `ShouldBindWith` .

You can also specify that specific fields are required. If a field is decorated with `binding:"required"` and has a empty value when binding, an error will be returned.

```go
// Binding from JSON
type Login struct {
	User     string `form:"user" json:"user" xml:"user"  binding:"required"`
	Password string `form:"password" json:"password" xml:"password" 
binding:"required"`
}

func main() {
	router := gin.Default()

	// Example for binding JSON ({"user": "manu", "password": "123"})
	router.POST("/loginJSON", func(c *gin.Context) {
		var json Login
		if err := c.ShouldBindJSON(&json); err != nil {
			c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
			return
		}

		if json.User != "manu" || json.Password != "123" {
			c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
			return
		}

		c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
	})
```

```
    // Example for binding XML (
    //  <?xml version="1.0" encoding="UTF-8"?>
    //  <root>
    //          <user>manu</user>
    //          <password>123</password>
    //  </root>)
    router.POST("/loginXML", func(c *gin.Context) {
        var xml Login
        if err := c.ShouldBindXML(&xml); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }

        if xml.User != "manu" || xml.Password != "123" {
            c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
            return
        }

        c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
    })

    // Example for binding a HTML form (user=manu&password=123)
    router.POST("/loginForm", func(c *gin.Context) {
        var form Login
        // This will infer what binder to use depending on the content-type header.
        if err := c.ShouldBind(&form); err != nil {
            c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
            return
        }

        if form.User != "manu" || form.Password != "123" {
            c.JSON(http.StatusUnauthorized, gin.H{"status": "unauthorized"})
            return
        }

        c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
    })

    // Listen and serve on 0.0.0.0:8080
    router.Run(":8080")
}
```

Sample request

```
$ curl -v -X POST \
  http://localhost:8080/loginJSON \
  -H 'content-type: application/json' \
  -d '{ "user": "manu" }'
> POST /loginJSON HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.51.0
> Accept: */*
> content-type: application/json
> Content-Length: 18
>
* upload completely sent off: 18 out of 18 bytes
< HTTP/1.1 400 Bad Request
< Content-Type: application/json; charset=utf-8
< Date: Fri, 04 Aug 2017 03:51:31 GMT
< Content-Length: 100
<
{"error":"Key: 'Login.Password' Error:Field validation for 'Password' failed on the
'required' tag"}
```

Skip validate

When running the above example using the above the `curl` command, it returns error. Because the example use `binding:"required"` for `Password`. If use `binding:"-"` for `Password`, then it will not return error when running the above example again.

Custom Validators

It is also possible to register custom validators. See the example code.

```go
package main

import (
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/gin-gonic/gin/binding"
    "github.com/go-playground/validator/v10"
)

// Booking contains binded and validated data.
type Booking struct {
    CheckIn  time.Time `form:"check_in" binding:"required,bookabledate"
time_format:"2006-01-02"`
    CheckOut time.Time `form:"check_out" binding:"required,gtfield=CheckIn"
time_format:"2006-01-02"`
}

var bookableDate validator.Func = func(fl validator.FieldLevel) bool {
    date, ok := fl.Field().Interface().(time.Time)
    if ok {
        today := time.Now()
        if today.After(date) {
            return false
        }
    }
    return true
}
```

```
func main() {
	route := gin.Default()

	if v, ok := binding.Validator.Engine().(*validator.Validate); ok {
		v.RegisterValidation("bookabledate", bookableDate)
	}

	route.GET("/bookable", getBookable)
	route.Run(":8085")
}

func getBookable(c *gin.Context) {
	var b Booking
	if err := c.ShouldBindWith(&b, binding.Query); err == nil {
		c.JSON(http.StatusOK, gin.H{"message": "Booking dates are valid!"})
	} else {
		c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
	}
}
```

```
$ curl "localhost:8085/bookable?check_in=2030-04-16&check_out=2030-04-17"
{"message":"Booking dates are valid!"}

$ curl "localhost:8085/bookable?check_in=2030-03-10&check_out=2030-03-09"
{"error":"Key: 'Booking.CheckOut' Error:Field validation for 'CheckOut' failed on the 'gtfield' tag"}

$ curl "localhost:8085/bookable?check_in=2000-03-09&check_out=2000-03-10"
{"error":"Key: 'Booking.CheckIn' Error:Field validation for 'CheckIn' failed on the 'bookabledate' tag"}%
```

Struct level validations can also be registered this way. See the struct-lvl-validation example to learn more.

Only Bind Query String

`ShouldBindQuery` function only binds the query params and not the post data. See the detail information.

```
package main

import (
	"log"

	"github.com/gin-gonic/gin"
)

type Person struct {
	Name    string `form:"name"`
	Address string `form:"address"`
}
```

```go
func main() {
	route := gin.Default()
	route.Any("/testing", startPage)
	route.Run(":8085")
}

func startPage(c *gin.Context) {
	var person Person
	if c.ShouldBindQuery(&person) == nil {
		log.Println("====== Only Bind By Query String ======")
		log.Println(person.Name)
		log.Println(person.Address)
	}
	c.String(200, "Success")
}
```

Bind Query String or Post Data

See the detail information.

```go
package main

import (
	"log"
	"time"

	"github.com/gin-gonic/gin"
)

type Person struct {
		Name       string    `form:"name"`
		Address    string    `form:"address"`
		Birthday   time.Time `form:"birthday" time_format:"2006-01-02" time_utc:"1"`
		CreateTime time.Time `form:"createTime" time_format:"unixNano"`
		UnixTime   time.Time `form:"unixTime" time_format:"unix"`
}

func main() {
	route := gin.Default()
	route.GET("/testing", startPage)
	route.Run(":8085")
}

func startPage(c *gin.Context) {
	var person Person
	// If `GET`, only `Form` binding engine (`query`) used.
	// If `POST`, first checks the `content-type` for `JSON` or `XML`, then uses
`Form` (`form-data`).
	// See more at https://github.com/gin-
gonic/gin/blob/master/binding/binding.go#L88
		if c.ShouldBind(&person) == nil {
```

```
                log.Println(person.Name)
                log.Println(person.Address)
                log.Println(person.Birthday)
                log.Println(person.CreateTime)
                log.Println(person.UnixTime)
        }

    c.String(200, "Success")
}
```

Test it with:

```
$ curl -X GET "localhost:8085/testing?name=appleboy&address=xyz&birthday=1992-03-
15&createTime=1562400033000000123&unixTime=1562400033"
```

Bind Uri

See the detail information.

```
package main

import "github.com/gin-gonic/gin"

type Person struct {
    ID string `uri:"id" binding:"required,uuid"`
    Name string `uri:"name" binding:"required"`
}

func main() {
    route := gin.Default()
    route.GET("/:name/:id", func(c *gin.Context) {
        var person Person
        if err := c.ShouldBindUri(&person); err != nil {
            c.JSON(400, gin.H{"msg": err.Error()})
            return
        }
        c.JSON(200, gin.H{"name": person.Name, "uuid": person.ID})
    })
    route.Run(":8088")
}
```

Test it with:

```
$ curl -v localhost:8088/thinkerou/987fbc97-4bed-5078-9f07-9141ba07c9f3
$ curl -v localhost:8088/thinkerou/not-uuid
```

Bind Header

```go
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
)

type testHeader struct {
    Rate   int    `header:"Rate"`
    Domain string `header:"Domain"`
}

func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        h := testHeader{}

        if err := c.ShouldBindHeader(&h); err != nil {
            c.JSON(200, err)
        }

        fmt.Printf("%#v\n", h)
        c.JSON(200, gin.H{"Rate": h.Rate, "Domain": h.Domain})
    })

    r.Run()

// client
// curl -H "rate:300" -H "domain:music" 127.0.0.1:8080/
// output
// {"Domain":"music","Rate":300}
}
```

Bind HTML checkboxes

See the detail information

main.go

```go
...

type myForm struct {
    Colors []string `form:"colors[]"`
}

...

func formHandler(c *gin.Context) {
    var fakeForm myForm
    c.ShouldBind(&fakeForm)
    c.JSON(200, gin.H{"color": fakeForm.Colors})
}

...
```

form.html

```html
<form action="/" method="POST">
    <p>Check some colors</p>
    <label for="red">Red</label>
    <input type="checkbox" name="colors[]" value="red" id="red">
    <label for="green">Green</label>
    <input type="checkbox" name="colors[]" value="green" id="green">
    <label for="blue">Blue</label>
    <input type="checkbox" name="colors[]" value="blue" id="blue">
    <input type="submit">
</form>
```

result:

```
{"color":["red","green","blue"]}
```

Multipart/Urlencoded binding

```go
type ProfileForm struct {
    Name   string                `form:"name" binding:"required"`
    Avatar *multipart.FileHeader `form:"avatar" binding:"required"`

    // or for multiple files
    // Avatars []*multipart.FileHeader `form:"avatar" binding:"required"`
}

func main() {
    router := gin.Default()
    router.POST("/profile", func(c *gin.Context) {
        // you can bind multipart form with explicit binding declaration:
        // c.ShouldBindWith(&form, binding.Form)
        // or you can simply use autobinding with ShouldBind method:
        var form ProfileForm
        // in this case proper binding will be automatically selected
        if err := c.ShouldBind(&form); err != nil {
            c.String(http.StatusBadRequest, "bad request")
            return
        }

        err := c.SaveUploadedFile(form.Avatar, form.Avatar.Filename)
        if err != nil {
            c.String(http.StatusInternalServerError, "unknown error")
            return
        }
        // db.Save(&form)

        c.String(http.StatusOK, "ok")
    })
    router.Run(":8080")
}
```

Test it with:

```
$ curl -X POST -v --form name=user --form "avatar=@./avatar.png"
http://localhost:8080/profile
```

XML, JSON, YAML and ProtoBuf rendering

```
func main() {
    r := gin.Default()

    // gin.H is a shortcut for map[string]interface{}
    r.GET("/someJSON", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/moreJSON", func(c *gin.Context) {
        // You also can use a struct
        var msg struct {
            Name    string `json:"user"`
            Message string
            Number  int
        }
        msg.Name = "Lena"
        msg.Message = "hey"
        msg.Number = 123
        // Note that msg.Name becomes "user" in the JSON
        // Will output  :   {"user": "Lena", "Message": "hey", "Number": 123}
        c.JSON(http.StatusOK, msg)
    })

    r.GET("/someXML", func(c *gin.Context) {
        c.XML(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/someYAML", func(c *gin.Context) {
        c.YAML(http.StatusOK, gin.H{"message": "hey", "status": http.StatusOK})
    })

    r.GET("/someProtoBuf", func(c *gin.Context) {
        reps := []int64{int64(1), int64(2)}
        label := "test"
        // The specific definition of protobuf is written in the
testdata/protoexample file.
```

```
        data := &protoexample.Test{
            Label: &label,
            Reps:  reps,
        }
        // Note that data becomes binary data in the response
        // Will output protoexample.Test protobuf serialized data
        c.ProtoBuf(http.StatusOK, data)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

SecureJSON

Using SecureJSON to prevent json hijacking. Default prepends `"while(1),"` to response body if the given struct is array values.

```
func main() {
    r := gin.Default()

    // You can also use your own secure json prefix
    // r.SecureJsonPrefix(")]}',\n")

    r.GET("/someJSON", func(c *gin.Context) {
        names := []string{"lena", "austin", "foo"}

        // Will output  :   while(1);["lena","austin","foo"]
        c.SecureJSON(http.StatusOK, names)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

JSONP

Using JSONP to request data from a server in a different domain. Add callback to response body if the query parameter callback exists.

```
func main() {
    r := gin.Default()

    r.GET("/JSONP", func(c *gin.Context) {
        data := gin.H{
            "foo": "bar",
        }

        //callback is x
        // Will output  :   x({\"foo\":\"bar\"})
        c.JSONP(http.StatusOK, data)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")

        // client
        // curl http://127.0.0.1:8080/JSONP?callback=x
}
```

AsciiJSON

Using AsciiJSON to Generates ASCII-only JSON with escaped non-ASCII characters.

```
func main() {
    r := gin.Default()

    r.GET("/someJSON", func(c *gin.Context) {
        data := gin.H{
            "lang": "GO语言",
            "tag":  "<br>",
        }

        // will output : {"lang":"GO\u8bed\u8a00","tag":"\u003cbr\u003e"}
        c.AsciiJSON(http.StatusOK, data)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

PureJSON

Normally, JSON replaces special HTML characters with their unicode entities, e.g. `<` becomes `\u003c`. If you want to encode such characters literally, you can use PureJSON instead. This feature is unavailable in Go 1.6 and lower.

```
func main() {
    r := gin.Default()

    // Serves unicode entities
    r.GET("/json", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "html": "<b>Hello, world!</b>",
        })
    })

    // Serves literal characters
    r.GET("/purejson", func(c *gin.Context) {
        c.PureJSON(200, gin.H{
            "html": "<b>Hello, world!</b>",
        })
    })
    // listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

Serving static files

```
func main() {
    router := gin.Default()
    router.Static("/assets", "./assets")
    router.StaticFS("/more_static", http.Dir("my_file_system"))
    router.StaticFile("/favicon.ico", "./resources/favicon.ico")
    router.StaticFileFS("/more_favicon.ico", "more_favicon.ico",
http.Dir("my_file_system"))

    // Listen and serve on 0.0.0.0:8080
    router.Run(":8080")
}
```

Serving data from file

```
func main() {
    router := gin.Default()

    router.GET("/local/file", func(c *gin.Context) {
        c.File("local/file.go")
    })

    var fs http.FileSystem = // ...
    router.GET("/fs/file", func(c *gin.Context) {
        c.FileFromFS("fs/file.go", fs)
    })
}
```

Serving data from reader

```
func main() {
    router := gin.Default()
    router.GET("/someDataFromReader", func(c *gin.Context) {
        response, err := http.Get("https://raw.githubusercontent.com/gin-
gonic/logo/master/color.png")
        if err != nil || response.StatusCode != http.StatusOK {
            c.Status(http.StatusServiceUnavailable)
            return
        }

        reader := response.Body
         defer reader.Close()
        contentLength := response.ContentLength
        contentType := response.Header.Get("Content-Type")

        extraHeaders := map[string]string{
            "Content-Disposition": `attachment; filename="gopher.png"`,
        }

        c.DataFromReader(http.StatusOK, contentLength, contentType, reader,
extraHeaders)
    })
    router.Run(":8080")
}
```

HTML rendering

Using LoadHTMLGlob() or LoadHTMLFiles()

```
func main() {
    router := gin.Default()
    router.LoadHTMLGlob("templates/*")
    //router.LoadHTMLFiles("templates/template1.html", "templates/template2.html")
    router.GET("/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.tmpl", gin.H{
            "title": "Main website",
        })
    })
    router.Run(":8080")
}
```

templates/index.tmpl

```
<html>
    <h1>
        {{ .title }}
    </h1>
</html>
```

Using templates with same name in different directories

```
func main() {
    router := gin.Default()
    router.LoadHTMLGlob("templates/**/*")
    router.GET("/posts/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "posts/index.tmpl", gin.H{
            "title": "Posts",
        })
    })
    router.GET("/users/index", func(c *gin.Context) {
        c.HTML(http.StatusOK, "users/index.tmpl", gin.H{
            "title": "Users",
        })
    })
    router.Run(":8080")
}
```

templates/posts/index.tmpl

```
{{ define "posts/index.tmpl" }}
<html><h1>
    {{ .title }}
</h1>
<p>Using posts/index.tmpl</p>
</html>
{{ end }}
```

templates/users/index.tmpl

```
{{ define "users/index.tmpl" }}
<html><h1>
    {{ .title }}
</h1>
<p>Using users/index.tmpl</p>
</html>
{{ end }}
```

Custom Template renderer

You can also use your own html template render

```go
import "html/template"

func main() {
    router := gin.Default()
    html := template.Must(template.ParseFiles("file1", "file2"))
    router.SetHTMLTemplate(html)
    router.Run(":8080")
}
```

Custom Delimiters

You may use custom delims

```go
    r := gin.Default()
    r.Delims("{[{", "}]}")
    r.LoadHTMLGlob("/path/to/templates")
```

Custom Template Funcs

See the detail example code.

main.go

```
import (
    "fmt"
    "html/template"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
)

func formatAsDate(t time.Time) string {
    year, month, day := t.Date()
    return fmt.Sprintf("%d/%02d/%02d", year, month, day)
}

func main() {
    router := gin.Default()
    router.Delims("{[{", "}]}")
    router.SetFuncMap(template.FuncMap{
        "formatAsDate": formatAsDate,
    })
    router.LoadHTMLFiles("./testdata/template/raw.tmpl")

    router.GET("/raw", func(c *gin.Context) {
        c.HTML(http.StatusOK, "raw.tmpl", gin.H{
            "now": time.Date(2017, 07, 01, 0, 0, 0, 0, time.UTC),
        })
    })

    router.Run(":8080")
}
```

raw.tmpl

```
Date: {[{.now | formatAsDate}]}
```

Result:

```
Date: 2017/07/01
```

Multitemplate

Gin allow by default use only one html.Template. Check a multitemplate render for using features like go 1.6 `block template`.

Redirects

Issuing a HTTP redirect is easy. Both internal and external locations are supported.

```
r.GET("/test", func(c *gin.Context) {
    c.Redirect(http.StatusMovedPermanently, "http://www.google.com/")
})
```

Issuing a HTTP redirect from POST. Refer to issue: [#444]

```
r.POST("/test", func(c *gin.Context) {
    c.Redirect(http.StatusFound, "/foo")
})
```

Issuing a Router redirect, use `HandleContext` like below.

```
r.GET("/test", func(c *gin.Context) {
    c.Request.URL.Path = "/test2"
    r.HandleContext(c)
})
r.GET("/test2", func(c *gin.Context) {
    c.JSON(200, gin.H{"hello": "world"})
})
```

Custom Middleware

```
func Logger() gin.HandlerFunc {
    return func(c *gin.Context) {
        t := time.Now()

        // Set example variable
        c.Set("example", "12345")

        // before request

        c.Next()

        // after request
        latency := time.Since(t)
        log.Print(latency)

        // access the status we are sending
        status := c.Writer.Status()
        log.Println(status)
    }
}

func main() {
    r := gin.New()
    r.Use(Logger())

    r.GET("/test", func(c *gin.Context) {
        example := c.MustGet("example").(string)

        // it would print: "12345"
```

```go
        log.Println(example)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

Using BasicAuth() middleware

```go
// simulate some private data
var secrets = gin.H{
    "foo":    gin.H{"email": "foo@bar.com", "phone": "123433"},
    "austin": gin.H{"email": "austin@example.com", "phone": "666"},
    "lena":   gin.H{"email": "lena@guapa.com", "phone": "523443"},
}

func main() {
    r := gin.Default()

    // Group using gin.BasicAuth() middleware
    // gin.Accounts is a shortcut for map[string]string
    authorized := r.Group("/admin", gin.BasicAuth(gin.Accounts{
        "foo":    "bar",
        "austin": "1234",
        "lena":   "hello2",
        "manu":   "4321",
    }))

    // /admin/secrets endpoint
    // hit "localhost:8080/admin/secrets
    authorized.GET("/secrets", func(c *gin.Context) {
        // get user, it was set by the BasicAuth middleware
        user := c.MustGet(gin.AuthUserKey).(string)
        if secret, ok := secrets[user]; ok {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": secret})
        } else {
            c.JSON(http.StatusOK, gin.H{"user": user, "secret": "NO SECRET :("})
        }
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

Goroutines inside a middleware

When starting new Goroutines inside a middleware or handler, you **SHOULD NOT** use the original context inside it, you have to use a read-only copy.

```
func main() {
    r := gin.Default()

    r.GET("/long_async", func(c *gin.Context) {
        // create copy to be used inside the goroutine
        cCp := c.Copy()
        go func() {
            // simulate a long task with time.Sleep(). 5 seconds
            time.Sleep(5 * time.Second)

            // note that you are using the copied context "cCp", IMPORTANT
            log.Println("Done! in path " + cCp.Request.URL.Path)
        }()
    })

    r.GET("/long_sync", func(c *gin.Context) {
        // simulate a long task with time.Sleep(). 5 seconds
        time.Sleep(5 * time.Second)

        // since we are NOT using a goroutine, we do not have to copy the context
        log.Println("Done! in path " + c.Request.URL.Path)
    })

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

Custom HTTP configuration

Use `http.ListenAndServe()` directly, like this:

```
func main() {
    router := gin.Default()
    http.ListenAndServe(":8080", router)
}
```

or

```
func main() {
    router := gin.Default()

    s := &http.Server{
        Addr:           ":8080",
        Handler:        router,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }
    s.ListenAndServe()
}
```

Support Let's Encrypt

example for 1-line LetsEncrypt HTTPS servers.

```
package main

import (
    "log"

    "github.com/gin-gonic/autotls"
    "github.com/gin-gonic/gin"
)

func main() {
    r := gin.Default()

    // Ping handler
    r.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    log.Fatal(autotls.Run(r, "example1.com", "example2.com"))
}
```

example for custom autocert manager.

```
package main

import (
    "log"

    "github.com/gin-gonic/autotls"
    "github.com/gin-gonic/gin"
    "golang.org/x/crypto/acme/autocert"
)

func main() {
    r := gin.Default()

    // Ping handler
    r.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })

    m := autocert.Manager{
        Prompt:     autocert.AcceptTOS,
        HostPolicy: autocert.HostWhitelist("example1.com", "example2.com"),
        Cache:      autocert.DirCache("/var/www/.cache"),
    }
    log.Fatal(autotls.RunWithManager(r, &m))
}
```

Run multiple service using Gin

See the question and try the following example:

```go
package main

import (
    "log"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "golang.org/x/sync/errgroup"
)

var (
    g errgroup.Group
)

func router01() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 01",
            },
        )
    })

    return e
}

func router02() http.Handler {
    e := gin.New()
    e.Use(gin.Recovery())
    e.GET("/", func(c *gin.Context) {
        c.JSON(
            http.StatusOK,
            gin.H{
                "code":  http.StatusOK,
                "error": "Welcome server 02",
            },
        )
    })
```

```go
	return e
}

func main() {
	server01 := &http.Server{
		Addr:         ":8080",
		Handler:      router01(),
		ReadTimeout:  5 * time.Second,
		WriteTimeout: 10 * time.Second,
	}

	server02 := &http.Server{
		Addr:         ":8081",
		Handler:      router02(),
		ReadTimeout:  5 * time.Second,
		WriteTimeout: 10 * time.Second,
	}

	g.Go(func() error {
		err := server01.ListenAndServe()
		if err != nil && err != http.ErrServerClosed {
			log.Fatal(err)
		}
		return err
	})

	g.Go(func() error {
		err := server02.ListenAndServe()
		if err != nil && err != http.ErrServerClosed {
			log.Fatal(err)
		}
		return err
	})

	if err := g.Wait(); err != nil {
		log.Fatal(err)
	}
}
```

Graceful shutdown or restart

There are a few approaches you can use to perform a graceful shutdown or restart. You can make use of third-party packages specifically built for that, or you can manually do the same with the functions and methods from the built-in packages.

Third-party packages

We can use fvbock/endless to replace the default `ListenAndServe`. Refer to issue #296 for more details.

```go
router := gin.Default()
router.GET("/", handler)
```

```
// [...]
endless.ListenAndServe(":4242", router)
```

Alternatives:

- manners: A polite Go HTTP server that shuts down gracefully.
- graceful: Graceful is a Go package enabling graceful shutdown of an http.Handler server.
- grace: Graceful restart & zero downtime deploy for Go servers.

Manually

In case you are using Go 1.8 or a later version, you may not need to use those libraries. Consider using `http.Server`'s built-in Shutdown() method for graceful shutdowns. The example below describes its usage, and we've got more examples using gin here.

```go
// +build go1.8

package main

import (
	"context"
	"log"
	"net/http"
	"os"
	"os/signal"
	"syscall"
	"time"

	"github.com/gin-gonic/gin"
)

func main() {
	router := gin.Default()
	router.GET("/", func(c *gin.Context) {
		time.Sleep(5 * time.Second)
		c.String(http.StatusOK, "Welcome Gin Server")
	})

	srv := &http.Server{
		Addr:    ":8080",
		Handler: router,
	}

	// Initializing the server in a goroutine so that
	// it won't block the graceful shutdown handling below
	go func() {
		if err := srv.ListenAndServe(); err != nil && errors.Is(err,
http.ErrServerClosed) {
			log.Printf("listen: %s\n", err)
		}
	}()
```

```
    // Wait for interrupt signal to gracefully shutdown the server with
    // a timeout of 5 seconds.
    quit := make(chan os.Signal)
    // kill (no param) default send syscall.SIGTERM
    // kill -2 is syscall.SIGINT
    // kill -9 is syscall.SIGKILL but can't be caught, so don't need to add it
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
    <-quit
    log.Println("Shutting down server...")

    // The context is used to inform the server it has 5 seconds to finish
    // the request it is currently handling
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    if err := srv.Shutdown(ctx); err != nil {
        log.Fatal("Server forced to shutdown:", err)
    }

    log.Println("Server exiting")
}
```

Build a single binary with templates

You can build a server into a single binary containing templates by using go-assets.

```
func main() {
    r := gin.New()

    t, err := loadTemplate()
    if err != nil {
        panic(err)
    }
    r.SetHTMLTemplate(t)

    r.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "/html/index.tmpl",nil)
    })
    r.Run(":8080")
}

// loadTemplate loads templates embedded by go-assets-builder
func loadTemplate() (*template.Template, error) {
    t := template.New("")
    for name, file := range Assets.Files {
        defer file.Close()
        if file.IsDir() || !strings.HasSuffix(name, ".tmpl") {
            continue
        }
        h, err := ioutil.ReadAll(file)
        if err != nil {
            return nil, err
        }
        t, err = t.New(name).Parse(string(h))
        if err != nil {
            return nil, err
        }
    }
    return t, nil
}
```

See a complete example in the `https://github.com/gin-gonic/examples/tree/master/assets-in-binary` directory.

Bind form-data request with custom struct

The follow example using custom struct:

```
type StructA struct {
    FieldA string `form:"field_a"`
}

type StructB struct {
    NestedStruct StructA
    FieldB string `form:"field_b"`
}

type StructC struct {
    NestedStructPointer *StructA
    FieldC string `form:"field_c"`
}

type StructD struct {
    NestedAnonyStruct struct {
        FieldX string `form:"field_x"`
    }
    FieldD string `form:"field_d"`
}

func GetDataB(c *gin.Context) {
    var b StructB
    c.Bind(&b)
    c.JSON(200, gin.H{
        "a": b.NestedStruct,
        "b": b.FieldB,
    })
}

func GetDataC(c *gin.Context) {
    var b StructC
    c.Bind(&b)
    c.JSON(200, gin.H{
        "a": b.NestedStructPointer,
        "c": b.FieldC,
    })
}

func GetDataD(c *gin.Context) {
    var b StructD
    c.Bind(&b)
    c.JSON(200, gin.H{
        "x": b.NestedAnonyStruct,
        "d": b.FieldD,
    })
}

func main() {
    r := gin.Default()
    r.GET("/getb", GetDataB)
    r.GET("/getc", GetDataC)
    r.GET("/getd", GetDataD)

    r.Run()
}
```

Using the command `curl` command result:

```
$ curl "http://localhost:8080/getb?field_a=hello&field_b=world"
{"a":{"FieldA":"hello"},"b":"world"}
$ curl "http://localhost:8080/getc?field_a=hello&field_c=world"
{"a":{"FieldA":"hello"},"c":"world"}
$ curl "http://localhost:8080/getd?field_x=hello&field_d=world"
{"d":"world","x":{"FieldX":"hello"}}
```

Try to bind body into different structs

The normal methods for binding request body consumes `c.Request.Body` and they cannot be called multiple times.

```
type formA struct {
  Foo string `json:"foo" xml:"foo" binding:"required"`
}

type formB struct {
  Bar string `json:"bar" xml:"bar" binding:"required"`
}

func SomeHandler(c *gin.Context) {
  objA := formA{}
  objB := formB{}
  // This c.ShouldBind consumes c.Request.Body and it cannot be reused.
  if errA := c.ShouldBind(&objA); errA == nil {
    c.String(http.StatusOK, `the body should be formA`)
    // Always an error is occurred by this because c.Request.Body is EOF now.
  } else if errB := c.ShouldBind(&objB); errB == nil {
    c.String(http.StatusOK, `the body should be formB`)
  } else {
    ...
  }
}
```

For this, you can use `c.ShouldBindBodyWith`.

```
func SomeHandler(c *gin.Context) {
  objA := formA{}
  objB := formB{}
  // This reads c.Request.Body and stores the result into the context.
  if errA := c.ShouldBindBodyWith(&objA, binding.Form); errA == nil {
    c.String(http.StatusOK, `the body should be formA`)
  // At this time, it reuses body stored in the context.
  } else if errB := c.ShouldBindBodyWith(&objB, binding.JSON); errB == nil {
    c.String(http.StatusOK, `the body should be formB JSON`)
  // And it can accepts other formats
  } else if errB2 := c.ShouldBindBodyWith(&objB, binding.XML); errB2 == nil {
    c.String(http.StatusOK, `the body should be formB XML`)
  } else {
    ...
  }
}
```

- `c.ShouldBindBodyWith` stores body into the context before binding. This has a slight impact to performance, so you should not use this method if you are enough to call binding at once.
- This feature is only needed for some formats -- `JSON`, `XML`, `MsgPack`, `ProtoBuf`. For other formats, `Query`, `Form`, `FormPost`, `FormMultipart`, can be called by `c.ShouldBind()` multiple times without any damage to performance (See #1341).

Bind form-data request with custom struct and custom tag

```
const (
    customerTag = "url"
    defaultMemory = 32 << 20
)

type customerBinding struct {}

func (customerBinding) Name() string {
    return "form"
}

func (customerBinding) Bind(req *http.Request, obj interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }
```

```go
    if err := req.ParseMultipartForm(defaultMemory); err != nil {
        if err != http.ErrNotMultipart {
            return err
        }
    }
    if err := binding.MapFormWithTag(obj, req.Form, customerTag); err != nil {
        return err
    }
    return validate(obj)
}

func validate(obj interface{}) error {
    if binding.Validator == nil {
        return nil
    }
    return binding.Validator.ValidateStruct(obj)
}

// Now we can do this!!!
// FormA is a external type that we can't modify it's tag
type FormA struct {
    FieldA string `url:"field_a"`
}

func ListHandler(s *Service) func(ctx *gin.Context) {
    return func(ctx *gin.Context) {
        var urlBinding = customerBinding{}
        var opt FormA
        err := ctx.MustBindWith(&opt, urlBinding)
        if err != nil {
            ...
        }
        ...
    }
}
```

http2 server push

http.Pusher is supported only **go1.8+**. See the golang blog for detail information.

```go
package main

import (
    "html/template"
    "log"

    "github.com/gin-gonic/gin"
)

var html = template.Must(template.New("https").Parse(`
<html>
<head>
  <title>Https Test</title>
  <script src="/assets/app.js"></script>
</head>
<body>
  <h1 style="color:red;">Welcome, Ginner!</h1>
</body>
</html>
`))

func main() {
	r := gin.Default()
	r.Static("/assets", "./assets")
	r.SetHTMLTemplate(html)

	r.GET("/", func(c *gin.Context) {
		if pusher := c.Writer.Pusher(); pusher != nil {
			// use pusher.Push() to do server push
			if err := pusher.Push("/assets/app.js", nil); err != nil {
				log.Printf("Failed to push: %v", err)
			}
		}
		c.HTML(200, "https", gin.H{
			"status": "success",
		})
	})

	// Listen and Server in https://127.0.0.1:8080
	r.RunTLS(":8080", "./testdata/server.pem", "./testdata/server.key")
}
```

Define format for the log of routes

The default log of routes is:

```
[GIN-debug] POST   /foo                      --> main.main.func1 (3 handlers)
[GIN-debug] GET    /bar                      --> main.main.func2 (3 handlers)
[GIN-debug] GET    /status                   --> main.main.func3 (3 handlers)
```

If you want to log this information in given format (e.g. JSON, key values or something else), then you can define this format with `gin.DebugPrintRouteFunc`. In the example below, we log all routes with standard log package but you can use another log tools that suits of your needs.

```
import (
	"log"
	"net/http"

	"github.com/gin-gonic/gin"
)

func main() {
```

```go
    r := gin.Default()
    gin.DebugPrintRouteFunc = func(httpMethod, absolutePath, handlerName string,
nuHandlers int) {
        log.Printf("endpoint %v %v %v %v\n", httpMethod, absolutePath, handlerName,
nuHandlers)
    }

    r.POST("/foo", func(c *gin.Context) {
        c.JSON(http.StatusOK, "foo")
    })

    r.GET("/bar", func(c *gin.Context) {
        c.JSON(http.StatusOK, "bar")
    })

    r.GET("/status", func(c *gin.Context) {
        c.JSON(http.StatusOK, "ok")
    })

    // Listen and Server in http://0.0.0.0:8080
    r.Run()
}
```

Set and get a cookie

```go
import (
    "fmt"

    "github.com/gin-gonic/gin"
)

func main() {

    router := gin.Default()

    router.GET("/cookie", func(c *gin.Context) {

        cookie, err := c.Cookie("gin_cookie")

        if err != nil {
            cookie = "NotSet"
            c.SetCookie("gin_cookie", "test", 3600, "/", "localhost", false, true)
        }

        fmt.Printf("Cookie value: %s \n", cookie)
    })

    router.Run()
}
```

Don't trust all proxies

Gin lets you specify which headers to hold the real client IP (if any), as well as specifying which proxies (or direct clients) you trust to specify one of these headers.

Use function `SetTrustedProxies()` on your `gin.Engine` to specify network addresses or network CIDRs from where clients which their request headers related to client IP can be trusted. They can be IPv4 addresses, IPv4 CIDRs, IPv6 addresses or IPv6 CIDRs.

Attention: Gin trust all proxies by default if you don't specify a trusted proxy using the function above, **this is NOT safe**. At the same time, if you don't use any proxy, you can disable this feature by using `Engine.SetTrustedProxies(nil)`, then `Context.ClientIP()` will return the remote address directly to avoid some unnecessary computation.

```
import (
	"fmt"

	"github.com/gin-gonic/gin"
)

func main() {

	router := gin.Default()
	router.SetTrustedProxies([]string{"192.168.1.2"})

	router.GET("/", func(c *gin.Context) {
		// If the client is 192.168.1.2, use the X-Forwarded-For
		// header to deduce the original client IP from the trust-
		// worthy parts of that header.
		// Otherwise, simply return the direct client IP
		fmt.Printf("ClientIP: %s\n", c.ClientIP())
	})
	router.Run()
}
```

Notice: If you are using a CDN service, you can set the `Engine.TrustedPlatform` to skip TrustedProxies check, it has a higher priority than TrustedProxies. Look at the example below:

```
import (
	"fmt"

	"github.com/gin-gonic/gin"
)

func main() {

	router := gin.Default()
	// Use predefined header gin.PlatformXXX
	router.TrustedPlatform = gin.PlatformGoogleAppEngine
	// Or set your own trusted request header for another trusted proxy service
	// Don't set it to any suspect request header, it's unsafe
```

```
    router.TrustedPlatform = "X-CDN-IP"

    router.GET("/", func(c *gin.Context) {
        // If you set TrustedPlatform, ClientIP() will resolve the
        // corresponding header and return IP directly
        fmt.Printf("ClientIP: %s\n", c.ClientIP())
    })
    router.Run()
}
```

Testing

The `net/http/httptest` package is preferable way for HTTP testing.

```go
package main

func setupRouter() *gin.Engine {
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.String(200, "pong")
    })
    return r
}

func main() {
    r := setupRouter()
    r.Run(":8080")
}
```

Test for code example above:

```go
package main

import (
    "net/http"
    "net/http/httptest"
    "testing"

    "github.com/stretchr/testify/assert"
)

func TestPingRoute(t *testing.T) {
    router := setupRouter()

    w := httptest.NewRecorder()
    req, _ := http.NewRequest("GET", "/ping", nil)
    router.ServeHTTP(w, req)

    assert.Equal(t, 200, w.Code)
    assert.Equal(t, "pong", w.Body.String())
}
```

Users

Awesome project lists using Gin web framework.

- gorush: A push notification server written in Go.
- fnproject: The container native, cloud agnostic serverless platform.
- photoprism: Personal photo management powered by Go and Google TensorFlow.
- krakend: Ultra performant API Gateway with middlewares.
- picfit: An image resizing server written in Go.
- brigade: Event-based Scripting for Kubernetes.
- dkron: Distributed, fault tolerant job scheduling system.