Since its launch, Golang (Google's Go programming language) has become a powerful and popular option to write APIs and web services. This list compiles the seven most popular web frameworks in Go — statistically speaking.

You might also want to check out DeepSource's static analysis for Go, that detects 300+ bug risks, anti-patterns, and security vulnerabilities in your Go code.

1. Gin

Gin is an HTTP web framework that features a Martini-like API with much better performance -- up to 40 times faster. If you need smashing performance, get yourself some Gin.

Features

- Faster performance: Radix tree based routing, small memory foot print. No reflection. Predictable API performance.
- Middleware support: An incoming HTTP request can be handled by a chain of middlewares and the final action. For example: Logger, Authorization, GZIP and finally post a message in the DB.
- Crash-free: Gin can catch a panic occurred during a HTTP request and recover it. This way, your server will be always available. As an example - it's also possible to report this panic to Sentry!
- JSON validation: Gin can parse and validate the JSON of a request - for example,checking the existence of required values.

- Routes grouping: Organize your routes better. Authorization required vs non required, different API versions... In addition, the groups can be nested unlimitedly without degrading performance.
- Error management: Gin provides a convenient way to collect all the errors occurred during a HTTP request. Eventually, a middleware can write them to a log file, to a database and send them through the network.
- Rendering built-in: Gin provides a easy to use API for JSON, XML and HTML rendering.
- Extendable: Creating a new middleware is so easy, just check out the sample codes.

Installation

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

Note: Go v1.10+ is required.

Hello world example

```
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")
}
```

Source: github.com/gin-gonic/gin

Documentation: gin-gonic.com/docs/

2. Beego

BEEGO

Beego is used for rapid development of RESTful APIs, web apps and backend services. It is inspired by Tornado, Sinatra and Flask. Beego has some Go-specific features such as interfaces and struct embedding.

Features

- With RESTful support, MVC model, and use bee tool to build your apps quickly with features including code hot compile, automated testing, and automated packing and deploying.
- With intelligent routing and monitoring, it's able to monitor your QPS, memory and CPU usages, and goroutine status. It provides you full control of your online apps.
- With powerful built-in modules including session control, caching, logging, configuration parsing, performance supervising, context handling, ORM supporting, and requests simulating. You get the powerful foundation for any type of applications.
- With native Go http package to handle the requests and the efficient concurrence of goroutine. Your beego applications can handle massive trafic as beego are doing in many productions.

Installation

```
go get -u github.com/astaxie/beego
```

Hello world example

```go
package main

import "github.com/astaxie/beego"

func main(){
    beego.Run()
}
```

Source: github.com/astaxie/beego

Documentation: beego.me/docs/intro/

3. Echo

Echo positions itself as high performance and minimalist web framework.

Features

- Optimized Router: Highly optimized HTTP router with zero dynamic memory allocation which smartly prioritizes routes.
- Scalability: Build robust and scalable RESTful API, easily organized into groups.
- Automatic TLS: Automatically install TLS certificates from Let's Encrypt.
- HTTP/2 support: HTTP/2 support improves speed and provides better user experience.
- Middlewares: Many built-in middleware to use, or define your own. Middleware can be set at root, group or route level.
- Data Binding: Data binding for HTTP request payload, including JSON, XML or form-data.

- Data Rendering: API to send variety of HTTP response, including JSON, XML, HTML, File, Attachment, Inline, Stream or Blob.
- Templating: Template rendering using any template engine.
- Extensibility: Customized central HTTP error handling. Easily extendable API.

Installation

```
go get -u github.com/labstack/echo
```

Hello world example

```go
package main

import (
  "net/http"
  "github.com/labstack/echo/v4"
  "github.com/labstack/echo/v4/middleware"
)

func main() {
  // Echo instance
  e := echo.New()

  // Middleware
  e.Use(middleware.Logger())
  e.Use(middleware.Recover())

  // Routes
  e.GET("/", hello)

  // Start server
  e.Logger.Fatal(e.Start(":1323"))
}

// Handler
func hello(c echo.Context) error {
  return c.String(http.StatusOK, "Hello, World!")
}
```

Source: github.com/labstack/echo

Documentation: echo.labstack.com/guide

4. Go kit

Go kit is a programming toolkit for building microservices (or elegant monoliths).

Features

- Operates in a heterogeneous SOA — expect to interact with mostly non-Go-kit services.
- RPC as the primary messaging pattern.
- Pluggable serialization and transport — not just JSON over HTTP.
- Operate within existing infrastructures — no mandates for specific tools or technologies.

Installation

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

Examples: godoc.org/github.com/go-kit/kit/examples

Source: github.com/go-kit/kit

Documentation: godoc.org/github.com/go-kit/kit

5. Fast HTTP

Fast HTTP package is tuned for high performance with zero memory allocations in hot paths. Performance wise, it is upto 10x faster than `net/http`

Features

- Optimized for speed: Easily handles more than 100K qps and more than 1M concurrent keep-alive connections on modern hardware.
- Optimized for low memory usage.
- Server provides many anti-DoS limits like concurrent connections per client IP, number of requests per connection and many more.
- Fasthttp API is designed with the ability to extend existing client and server implementations or to write custom client and server implementations from scratch.

Installation

```
go get -u github.com/valyala/fasthttp
```

Hello world example

```
package main

import (
	"flag"
	"fmt"
	"log"

	"github.com/valyala/fasthttp"
)

var (
	addr     = flag.String("addr", ":8080", "TCP address to listen to")
	compress = flag.Bool("compress", false, "Whether to enable transparent response compression")
)

func main() {
	flag.Parse()

	h := requestHandler
	if *compress {
		h = fasthttp.CompressHandler(h)
	}

	if err := fasthttp.ListenAndServe(*addr, h); err != nil {
		log.Fatalf("Error in ListenAndServe: %s", err)
	}
}

func requestHandler(ctx *fasthttp.RequestCtx) {
	fmt.Fprintf(ctx, "Hello, world!\n\n")
}
```

Source: github.com/valyala/fasthttp

Documentation: godoc.org/github.com/valyala/fasthttp

6. Mux

Mux (gorilla) implements a request router and dispatcher for matching incoming requests to their respective handler.

Features

- It implements the http.Handler interface so it is compatible with the standard http.ServeMux.
- Requests can be matched based on URL host, path, path prefix, schemes, header and query values, HTTP methods or using custom matchers.
- URL hosts, paths and query values can have variables with an optional regular expression.
- Registered URLs can be built, or “reversed”, which helps maintaining references to resources.
- Routes can be used as subrouters: nested routes are only tested if the parent route matches. This is useful to define groups of routes that share common conditions like a host, a path prefix or other repeated attributes. As a bonus, this optimizes request matching.

Installation

```
go get -u github.com/gorilla/mux
```

Hello world example

```
func main() {
    r := mux.NewRouter()
    r.HandleFunc("/articles", ArticlesHandler)
    http.Handle("/", r)
}

func    ArticlesCategoryHandler(w    http.ResponseWriter,    r
*http.Request) {
    vars := mux.Vars(r)
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "Category: %v\n", vars["category"])
}
```

Source: github.com/gorilla/mux

Documentation: gorillatoolkit.org/pkg/mux

7. HttpRouter

HttpRouter is a lightweight high performance HTTP request router (also called multiplexer or just mux for short).

In contrast to the default mux of Go’s `net/http` package, this router supports variables in the routing pattern and matches against the request method. It also scales better.
Features

- Only explicit matches: With other routers, like `http.ServeMux`, a requested URL path could match multiple patterns. Therefore they have some awkward pattern priority rules, like longest match or first registered, first matched. By design of this router, a request can only match exactly one or no route. As a result, there are also no unintended matches, which makes it great for SEO and improves the user experience.
- Stop caring about trailing slashes: Choose the URL style you like, the router automatically redirects the client if a trailing slash is missing or if there is one extra. Of course it only does so, if the new path has a handler. If you don’t like it, you can turn off this behavior.
- Path auto-correction: Besides detecting the missing or additional trailing slash at no extra cost, the router can also fix wrong cases and remove superfluous path elements (like ../ or //). Is CAPTAIN CAPS LOCK one of your users? HttpRouter can help him by making a case-insensitive look-up and redirecting him to the correct URL.
- Parameters in your routing pattern: Stop parsing the requested URL path, just give the path segment a name and

the router delivers the dynamic value to you. Because of the design of the router, path parameters are very cheap.

- Zero Garbage: The matching and dispatching process generates zero bytes of garbage. The only heap allocations that are made are building the slice of the key-value pairs for path parameters, and building new context and request objects (the latter only in the standard `Handler`/`HandlerFunc` API). In the 3-argument API, if the request path contains no parameters not a single heap allocation is necessary.
- Performance: Benchmarks speak for themselves.
- No more server crashes: You can set a Panic handler to deal with panics occurring during handling a HTTP request. The router then recovers and lets the PanicHandler log what happened and deliver a nice error page.
- Perfect for APIs: The router design encourages to build sensible, hierarchical RESTful APIs. Moreover it has built-in native support for `OPTIONS` requests and `405 Method Not Allowed` replies.

Installation

```
go get -u github.com/julienschmidt/httprouter
```

Hello world example

```
package main

import (
    "fmt"
    "net/http"
    "log"

    "github.com/julienschmidt/httprouter"
)
```

```go
func Index(w http.ResponseWriter, r *http.Request, _
httprouter.Params) {
    fmt.Fprint(w, "Welcome!\n")
}

func Hello(w http.ResponseWriter, r *http.Request, ps
httprouter.Params) {
    fmt.Fprintf(w, "hello, %s!\n", ps.ByName("name"))
}

func main() {
    router := httprouter.New()
    router.GET("/", Index)
    router.GET("/hello/:name", Hello)

    log.Fatal(http.ListenAndServe(":8080", router))
}
```

Source: github.com/julienschmidt/httprouter

Documentation: godoc.org/github.com/julienschmidt/httprouter