HttpRouter

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

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.

The router is optimized for high performance and a small memory footprint. It scales well even with very long paths and a large number of routes. A compressing dynamic trie (radix tree) structure is used for efficient matching.

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. In fact, the only heap allocations that are made, is by building the slice of the key-value pairs for path parameters. If the request path contains no parameters, not a single heap allocation is necessary.

Best Performance: Benchmarks speak for themselves. See below for technical details of the implementation.

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 builtin native support for OPTIONS requests and `405 Method Not Allowed` replies.

Of course you can also set **custom `NotFound` and `MethodNotAllowed` handlers** and **serve static files**.

Usage

This is just a quick introduction, view the GoDoc for details.

Let's start with a trivial example:

```
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
```

```
    "net/http"
    "log"
)

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.Param
    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))
}
```

Named parameters

As you can see, `:name` is a *named parameter*. The values are accessible via `httprouter.Params`, which is just a slice of `httprouter.Param`s. You can get the value of a parameter either by its index in the slice, or by using the `ByName(name)` method: `:name` can be retrived by `ByName("name")`.

Named parameters only match a single path segment:

```
Pattern: /user/:user

 /user/gordon              match
 /user/you                 match
 /user/gordon/profile      no match
 /user/                    no match
```

Note: Since this router has only explicit matches, you can not register static routes and parameters for the same path segment. For example you

can not register the patterns `/user/new` and `/user/:user` for the same request method at the same time. The routing of different request methods is independent from each other.

Catch-All parameters

The second type are *catch-all* parameters and have the form `*name`. Like the name suggests, they match everything. Therefore they must always be at the **end** of the pattern:

```
Pattern: /src/*filepath

 /src/                     match
 /src/somefile.go          match
 /src/subdir/somefile.go   match
```

How does it work?

The router relies on a tree structure which makes heavy use of *common prefixes*, it is basically a *compact* *prefix tree* (or just *Radix tree*). Nodes with a common prefix also share a common parent. Here is a short example what the routing tree for the `GET` request method could look like:

```
Priority   Path             Handle
9          \                *<1>
3          ├s               nil
2          │├earch\         *<2>
1          │└upport\        *<3>
2          ├blog\           *<4>
1          │    └:post      nil
1          │         └\     *<5>
2          ├about-us\       *<6>
1          │        └team\  *<7>
1          └contact\        *<8>
```

Every `*<num>` represents the memory address of a handler function (a

pointer). If you follow a path trough the tree from the root to the leaf, you get the complete route path, e.g `\blog\:post\`, where `:post` is just a placeholder (*parameter*) for an actual post name. Unlike hash-maps, a tree structure also allows us to use dynamic parts like the `:post` parameter, since we actually match against the routing patterns instead of just comparing hashes. As benchmarks show, this works very well and efficient.

Since URL paths have a hierarchical structure and make use only of a limited set of characters (byte values), it is very likely that there are a lot of common prefixes. This allows us to easily reduce the routing into ever smaller problems. Moreover the router manages a separate tree for every request method. For one thing it is more space efficient than holding a method->handle map in every single node, for another thing is also allows us to greatly reduce the routing problem before even starting the look-up in the prefix-tree.

For even better scalability, the child nodes on each tree level are ordered by priority, where the priority is just the number of handles registered in sub nodes (children, grandchildren, and so on..). This helps in two ways:

1. Nodes which are part of the most routing paths are evaluated first. This helps to make as much routes as possible to be reachable as fast as possible.
2. It is some sort of cost compensation. The longest reachable path (highest cost) can always be evaluated first. The following scheme visualizes the tree structure. Nodes are evaluated from top to bottom and from left to right.

```
├------------
├---------
├-----
├----
├--
├--
└-
```

Why doesn't this work with `http.Handler`?

It does! The router itself implements the `http.Handler` interface. Moreover the router provides convenient adapters for `http.Handler`s and `http.HandlerFunc`s which allows them to be used as a `httprouter.Handle` when registering a route. The only disadvantage is, that no parameter values can be retrieved when a `http.Handler` or `http.HandlerFunc` is used, since there is no efficient way to pass the values with the existing function parameters. Therefore `httprouter.Handle` has a third function parameter.

Just try it out for yourself, the usage of HttpRouter is very straightforward. The package is compact and minimalistic, but also probably one of the easiest routers to set up.

Where can I find Middleware *X*?

This package just provides a very efficient request router with a few extra features. The router is just a `http.Handler`, you can chain any http.Handler compatible middleware before the router, for example the Gorilla handlers. Or you could just write your own, it's very easy!

Alternatively, you could try a web framework based on HttpRouter.

Multi-domain / Sub-domains

Here is a quick example: Does your server serve multiple domains / hosts? You want to use sub-domains? Define a router per host!

```
// We need an object that implements the http.Handler interface.
// Therefore we need a type for which we implement the ServeHTTP metho
// We just use a map here, in which we map host names (with port) to h
type HostSwitch map[string]http.Handler

// Implement the ServerHTTP method on our new type
func (hs HostSwitch) ServeHTTP(w http.ResponseWriter, r *http.Request)
```

```
    // Check if a http.Handler is registered for the given host.
    // If yes, use it to handle the request.
    if handler := hs[r.Host]; handler != nil {
        handler.ServeHTTP(w, r)
    } else {
        // Handle host names for wich no handler is registered
        http.Error(w, "Forbidden", 403) // Or Redirect?
    }
}

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

    // Make a new HostSwitch and insert the router (our http handler)
    // for example.com and port 12345
    hs := make(HostSwitch)
    hs["example.com:12345"] = router

    // Use the HostSwitch to listen and serve on port 12345
    log.Fatal(http.ListenAndServe(":12345", hs))
}
```

Basic Authentication

Another quick example: Basic Authentication (RFC 2617) for handles:

```
package main

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

    "github.com/julienschmidt/httprouter"
)

func BasicAuth(h httprouter.Handle, requiredUser, requiredPassword str
    return func(w http.ResponseWriter, r *http.Request, ps httprouter.
```

```
        // Get the Basic Authentication credentials
        user, password, hasAuth := r.BasicAuth()

        if hasAuth && user == requiredUser && password == requiredPass
            // Delegate request to the given handle
            h(w, r, ps)
        } else {
            // Request Basic Authentication otherwise
            w.Header().Set("WWW-Authenticate", "Basic realm=Restricted
            http.Error(w, http.StatusText(http.StatusUnauthorized), ht
        }
    }
}

func Index(w http.ResponseWriter, r *http.Request, _ httprouter.Params
    fmt.Fprint(w, "Not protected!\n")
}

func Protected(w http.ResponseWriter, r *http.Request, _ httprouter.Pa
    fmt.Fprint(w, "Protected!\n")
}

func main() {
    user := "gordon"
    pass := "secret!"

    router := httprouter.New()
    router.GET("/", Index)
    router.GET("/protected/", BasicAuth(Protected, user, pass))

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

Chaining with the NotFound handler

NOTE: It might be required to set `Router.HandleMethodNotAllowed` to `false` to avoid problems.

You can use another `http.Handler`, for example another router, to handle requests which could not be matched by this router by using the `Router.NotFound` handler. This allows chaining.

Static files

The `NotFound` handler can for example be used to serve static files from the root path `/` (like an `index.html` file along with other assets):

```
// Serve static files from the ./public directory
router.NotFound = http.FileServer(http.Dir("public"))
```

But this approach sidesteps the strict core rules of this router to avoid routing problems. A cleaner approach is to use a distinct sub-path for serving files, like `/static/*filepath` or `/files/*filepath`.

Web Frameworks based on HttpRouter

If the HttpRouter is a bit too minimalistic for you, you might try one of the following more high-level 3rd-party web frameworks building upon the HttpRouter package:

- Ace: Blazing fast Go Web Framework
- api2go: A JSON API Implementation for Go
- Gin: Features a martini-like API with much better performance
- Goat: A minimalistic REST API server in Go
- Hikaru: Supports standalone and Google AppEngine
- Hitch: Hitch ties httprouter, httpcontext, and middleware up in a bow
- httpway: Simple middleware extension with context for httprouter and a server with gracefully shutdown support
- kami: A tiny web framework using x/net/context
- Medeina: Inspired by Ruby's Roda and Cuba
- Neko: A lightweight web application framework for Golang
- River: River is a simple and lightweight REST server
- Roxanna: An amalgamation of httprouter, better logging, and hot reload
- siesta: Composable HTTP handlers with contexts

- xmux: xmux is a httprouter fork on top of xhandler (net/context aware)