Better HTTP server routing in Go 1.22

An exciting proposal is expected to land in Go 1.22 - enhancing the pattern-matching capabilities of the default HTTP serving multiplexer in the `net/http` package.

The existing multiplexer (http.ServeMux) offers rudimentary path matching, but not much beyond that. This led to a cottage industry of 3rd party libraries to implement more powerful capabilities. I've explored these options in my *REST Servers in Go* series, in parts 1 and 2.

The new multiplexer in 1.22 is going to significantly bridge the gap from 3rd party packages by providing advanced matching. In this short post I'll provide a quick introduction to the new multiplexer (mux). I'll also revisit the example from the *REST Servers in Go* series and compare how the new stdlib mux fares against `gorilla/mux`.

Using the new mux

If you've ever used a 3rd party mux / router package for Go (like `gorilla/mux`), using the new standard mux is going to be straightforward and familiar. Start by reading its documentation - it's short and sweet.

Let's look at a couple of basic usage examples. Our first example demonstrates some of the new pattern matching capabilities of the mux:

```
package main

import (
  "fmt"
  "net/http"
)
```

```
func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("GET /path/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "got path\n")
  })

  mux.HandleFunc("/task/{id}/", func(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    fmt.Fprintf(w, "handling task with id=%v\n", id)
  })

  http.ListenAndServe("localhost:8090", mux)
}
```

Experienced Go programmers will notice two new features right away:

In the first handler, the HTTP method (`GET` in this case) is specified explicitly as part of the pattern. This means that this handler will only trigger for `GET` requests to paths beginning with `/path/`, not for other HTTP methods.

In the second handler, there's a wildcard in the second path component - `{id}`, something that wasn't supported before. The wildcard will match a single path component and the handler can then access the matched value through the `PathValue` method of the request.

Since Go 1.22 hasn't been released yet, I recommend running this sample with `gotip`. Please see the complete code sample with full instructions for running this. Let's take this server for a ride:

And in a separate terminal we can issue some `curl` calls to test it:

```
$ curl localhost:8090/what/
404 page not found

$ curl localhost:8090/path/
got path

$ curl -X POST localhost:8090/path/
Method Not Allowed

$ curl localhost:8090/task/f0cd2e/
handling task with id=f0cd2e
```

Note how the server rejects a `POST` request to `/path/`, while the (default for `curl`) `GET` request is allowed. Note also how the `id` wildcard gets assigned a value when the request matches. Once again, I encourage you to review the documentation of the new ServeMux. You'll learn about additional capabilities like matching trailing paths to a wildcard with `{id}...`, strict matching of a path end with `{$}`, and other rules.

Particular care in the proposal was given to potential conflicts between different patterns. Consider this setup:

```
mux := http.NewServeMux()
mux.HandleFunc("/task/{id}/status/", func(w http.ResponseWriter, r *http.Request) {
	id := r.PathValue("id")
	fmt.Fprintf(w, "handling task status with id=%v\n", id)
})
mux.HandleFunc("/task/0/{action}/", func(w http.ResponseWriter, r *http.Request) {
	action := r.PathValue("action")
	fmt.Fprintf(w, "handling task 0 with action=%v\n", action)
})
```

And suppose the server receives a request for `/task/0/status/` -- which handler should it go to? It matches both! Therefore, the new `ServeMux` documentation meticulously describes the *precedence rules* for patterns, along with potential conflicts. In case of a conflict, the registration panics. Indeed, for the example above we get something like:

```
panic: pattern "/task/0/{action}/" (registered at sample-conflict.go:14) conflicts with pattern "/task/{id}/status/"
(registered at sample-conflict.go:10):
/task/0/{action}/ and /task/{id}/status/ both match some paths, like "/task/0/status/".
But neither is more specific than the other.
/task/0/{action}/ matches "/task/0/action/", but /task/{id}/status/ doesn't.
/task/{id}/status/ matches "/task/id/status/", but /task/0/{action}/ doesn't.
```

The message is detailed and helpful. If we encounter conflicts in complex registration schemes (especially when patterns are registered in multiple places in the source code), such details will be much appreciated.

Redoing my task server with the new mux

The *REST Servers in Go* series implements a simple server for a task/todo-list application in Go, using several different approaches. Part 1 starts with a "vanilla" standard library approach, and Part 2 reimplements the same server using the gorilla/mux router.

Now is a great time to reimplement it once again, but with the enhanced mux from Go 1.22; it will be particularly interesting to compare the solution to the one using `gorilla/mux`.

The full code for this project is available here. Let's look at a few representative code samples, starting with the

pattern registration [1]:

```
mux := http.NewServeMux()
server := NewTaskServer()

mux.HandleFunc("POST /task/", server.createTaskHandler)
mux.HandleFunc("GET /task/", server.getAllTasksHandler)
mux.HandleFunc("DELETE /task/", server.deleteAllTasksHandler)
mux.HandleFunc("GET /task/{id}/", server.getTaskHandler)
mux.HandleFunc("DELETE /task/{id}/", server.deleteTaskHandler)
mux.HandleFunc("GET /tag/{tag}/", server.tagHandler)
mux.HandleFunc("GET /due/{year}/{month}/{day}/", server.dueHandler)
```

Just like in the `gorilla/mux` sample, here we use specific HTTP methods to route requests (with the same path) to different handlers; with the older `http.ServeMux`, such matchers had to go to the same handler, which would then decide what to do based on the method.

Let's also look at one of the handlers:

```
func (ts *taskServer) getTaskHandler(w http.ResponseWriter, req *http.Request) {
  log.Printf("handling get task at %s\n", req.URL.Path)

  id, err := strconv.Atoi(req.PathValue("id"))
  if err != nil {
    http.Error(w, "invalid id", http.StatusBadRequest)
    return
  }

  task, err := ts.store.GetTask(id)
  if err != nil {
    http.Error(w, err.Error(), http.StatusNotFound)
    return
  }

  renderJSON(w, task)
}
```

It extracts the ID value from `req.PathValue("id")`, similarly to the Gorilla approach; however, since we don't have a regexp specifying that `{id}` only matches integers, we have to pay attention to errors returned from `strconv.Atoi`.

All and all, the end result is remarkably similar to the solution that uses `gorilla/mux` from part 2. The handlers are much better separated than in the vanilla stdlib approach, because the mux now can do much more sophisticated routing, without leaving many of the routing decisions to the handlers themselves.

Conclusion

"Which router package should I use?" has always been a FAQ for beginner Go programmers. I believe the common answers to this question will shift after Go 1.22 is released, as many will find the new stdlib mux sufficient for their needs without resorting to 3rd party packages.

Others will stick to familiar 3rd party packages, and that's totally fine. Routers like `gorilla/mux` still provide more capabilities than the standard library; on top of it, many Go programmers opt for lightweight frameworks like Gin, which provide a router but also additional tools for building web backends.

All in all, this is certainly a positive change for all Go users. Making the standard library more capable is a net positive for the entire community, whether people use 3rd party packages or stick to just the standard library.

---

[1]	You may have noticed that these patterns aren't very strict w.r.t path parts that come after the part we care about (e.g. `/task/22/foobar`). This is in line with the rest of the series, but the new `http.ServeMux` makes it very easy to restrict the paths with a trailing `{$}` wildcard, if needed.