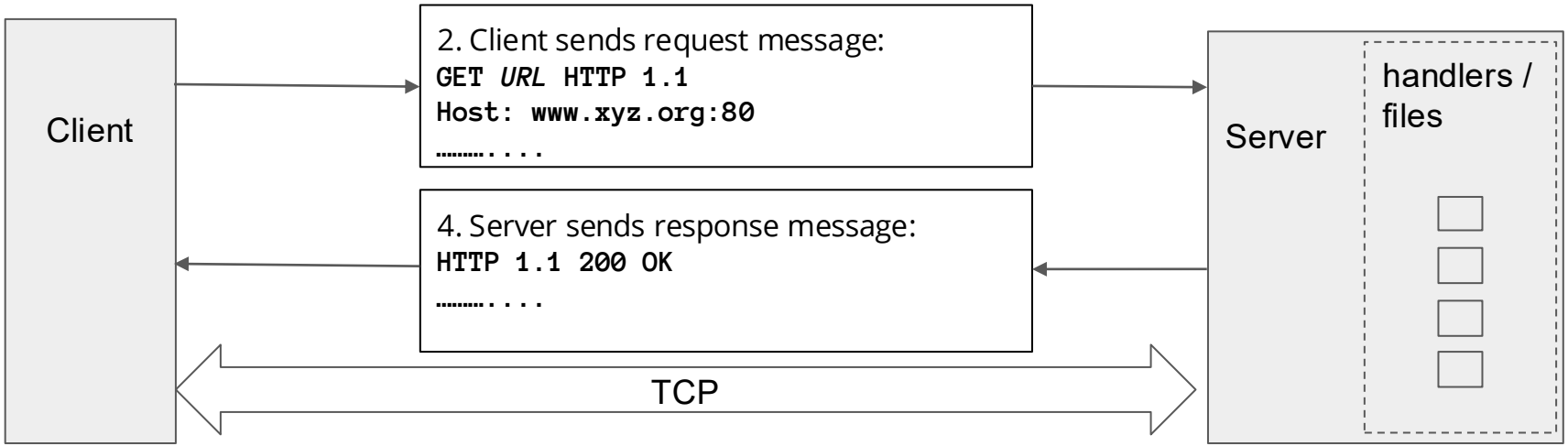
HTTP Router in Go

COS 316 Precept 6

HTTP Example

1. Client requests URL:
`http://www.xyz.org:80/path/file`

3. Server routes request to the appropriate handler/file



5. Client processes response

What is an HTTP router?

An HTTP router is a traffic director for web requests.

When a request comes in, it decides:

- which code should handle it, or
- whether to return 404

Example:

GET / hello -> run the hello handler

GET / users -> run the users handler

GET / unknown -> return 404

What is an HTTP request?

When a browser or client talks to a server, it sends a request

- Important pieces:
  - method: GET POST, etc
  - path: /hello, /users/cesar
  - maybe a query string: ?page=2

Example:

- GET /users/cesar?page=2
  - method = GET
  - path = /users/cesar
  - query string = page=2

What is an HTTP handler?

An HTTP handler is the code that handles the request

basic handler in Go: ServeHTTP(w http.ResponseWriter, r *http.Request)

r is the incoming request

w is where you write the response

So ServeHTTP(w, r) means: here is one request r; write the response to w.

A router is also a handler.

That means: router also has a ServeHTTP(w, r) method.

But the router usually does not generate the final page itself.

Instead, it does this:

inspect the request

find the correct handler

call that handler

So, the router's ServeHTTP is like a dispatcher.

A simple picture of control flow

Imagine a request comes in:

```
GET /hello
```

Then the flow is:

```
request arrives
    ↓
router ServeHTTP(w, r)
    ↓
router checks method/path
    ↓
router finds hello handler
    ↓
hello handler ServeHTTP(w, r)
    ↓
response sent back
```

If the router cannot find a match:

```
request arrives
    ↓
router ServeHTTP(w, r)
    ↓
no matching route
    ↓
404
```

What information does the router look at?

At first, only focus on these two:

- `r.Method`
- `r.URL.Path`

Example:

```
GET /hello
```

Then:

- `r.Method` is `"GET"`
- `r.URL.Path` is `"/hello"`

The router often uses those two to decide what to do.

More about path (important for assignment 2)

A path is a concrete string

Examples of request paths:
`/hello`
`/users`
`/users/cesar`
`/users/cesar/recent`
These are **actual paths** from actual requests.
So `/users/cesar` is one concrete path.

More about path (important for assignment 2)

But.... a router does not only store concrete paths

If we only stored exact paths, we would need one route for every user:

```
/users/alice
/users/bob
/users/cesar
/users/david
```

That would be very inconvenient.

So instead of only concrete paths, routers often use **patterns**.

Route Pattern

A route pattern is like a template for matching many concrete paths.

Example:
`/users/:user`
This is not one exact path.
It is a **pattern** that can match many paths, such as:
`/users/alice`
`/users/bob`
`/users/cesar`

“A route pattern describes a family of paths.”

Route Pattern

A pattern has **fixed parts** and **variable parts**
Look at this pattern:
`/users/:user`
It has two different kinds of pieces:
•fixed part: `users`
•variable part: `:user`

The fixed part must match **exactly**.
The variable part can match **different values**.
This is the main idea behind captures.

Captures

What does `:user` mean?
The piece `:user` means:
match one path segment here, and save it under the name `user`.

For example, if the pattern is:
`/users/:user`
and the request path is:
`/users/cesar`
then `:user` matches `cesar`.
And the router captures:
`user = cesar`

Captures

“Capture” means “remember the value”

A capture does two things:
1.it helps the pattern match the path,
2.it remembers the matched value.

So `:user` does not just mean “anything goes here.”
It means:
anything goes here, and I want to remember what it was.

Captures

Compare exact match and capture match

Exact segment
Pattern:
`/users/recent`
Path:
`/users/recent`
Here both segments must match exactly.

Capture segment
Pattern:
`/users/:user`
Path:
`/users/cesar`
Here:
- `users` matches exactly
- `:user` captures `cesar`

Path Segments

Paths are usually compared segment by segment.
For example:
`/users/cesar/recent`
can be thought of as segments:
`users`
`cesar`
`recent`
And the pattern:
`/users/:user/recent`
can be thought of as:
`users`
`:user`
`recent`
Then the router compares them one by one.

Example

Pattern:
`/users/:user/recent`
Path:
`/users/cesar/recent`

Compare segment by segment:
- `users` matches `users`
- `:user` captures `cesar`
- `recent` matches `recent`

So the whole pattern matches.
And the captured value is:
`user = cesar`

Captures

A capture matches only one segment

`:user` matches only **one** path segment.
So in:
`/users/:user`
the `:user` part can match:
`cesar`
`alice`
`bob`
but not something with another slash in it.

“A capture fills one slot in the path.”

Captures

Captures can appear in different positions
A capture does not have to be at the end.

Example:
`/teams/:team/members`
can match:
`/teams/tigers/members`
and capture:
`team = tigers`

“A capture can appear anywhere in the pattern.”

Captures

Multiple captures

Example:
`/users/:user/posts/:post`
Path:
`/users/cesar/posts/17`
Compare:
- `users` matches `users`
- `:user` captures `cesar`
- `posts` matches `posts`
- `:post` captures `17`

So we get:
```
user = cesar
post = 17
```
“A pattern can capture several values.”

```go
func main() {
    pattern := "/users/:user/posts/:post"
    path := "/users/cesar/posts/17"

    pSegs := strings.Split(strings.Trim(pattern, "/"), "/")
    pathSegs := strings.Split(strings.Trim(path, "/"), "/")

    captures := map[string][]string{}

    for i := 0; i < len(pSegs); i++ {
        ps := pSegs[i]
        rs := pathSegs[i]

        if strings.HasPrefix(ps, ":") {       // capture match, user:id is not a valid capture
            key := ps[1:]
            captures[key] = append(captures[key], rs)
        } else if ps != rs {          // exact match
            fmt.Println("no match")
            return
        }
    }
    fmt.Println("matched")
}
```