Using Go for Web Application

Lecture 03

Learning Outcomes

After completing this lesson you should be able to

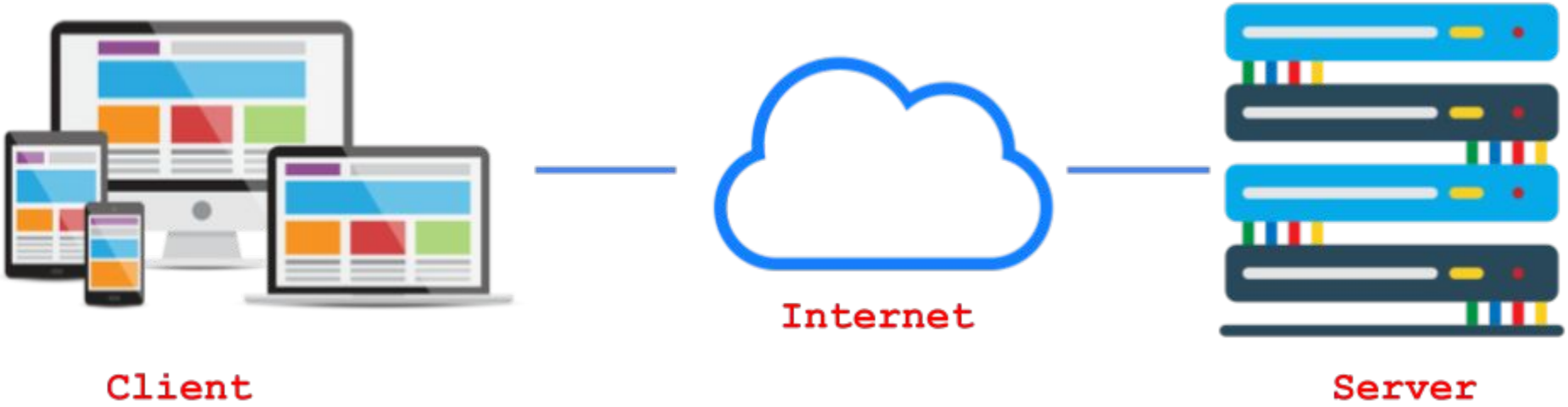
- Explain **how the web works**
- Explain the **components involved when you send a web request** to a web server across the internet
- Write **basic Go web server program** and explain the parts involved such as `http` and `template` packages; `HandleFunc`, `multiplexer`, `ListenAndServe`, **FileServer**

How the web works

What components are involved when you make a web request?

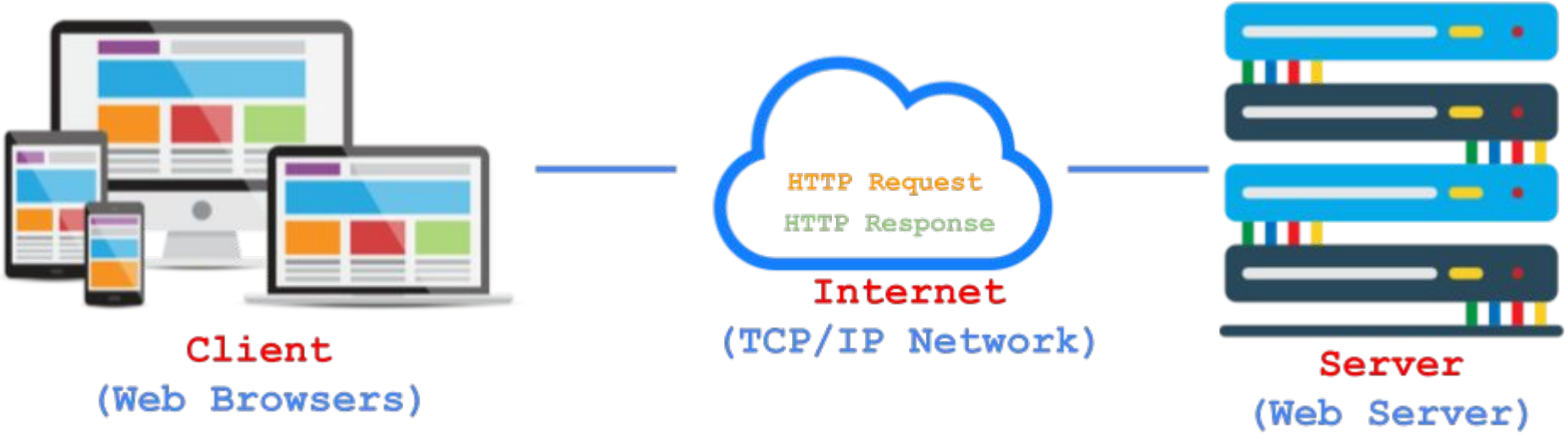
How the web works

Components Involved



How the web works

Components Involved



How the web works

Components Involved

Clients: Initiate HTTP requests

Server: Response to HTTP requests with HTTP response

Internet: Allows you to send and receive data on the web

TCP/IP: communication protocols that define how data should travel across the web

How the web works

Components Involved

DNS Servers: Special servers that match up a web address you type into your browser (like “www.google.com") to the website's real (IP) address.

When you type a web address in your browser, the browser looks at the DNS to find the website's real address before it can retrieve the website

HTTP: Hypertext Transfer Protocol is an application protocol that defines a language for clients and servers to speak to each other

How the web works

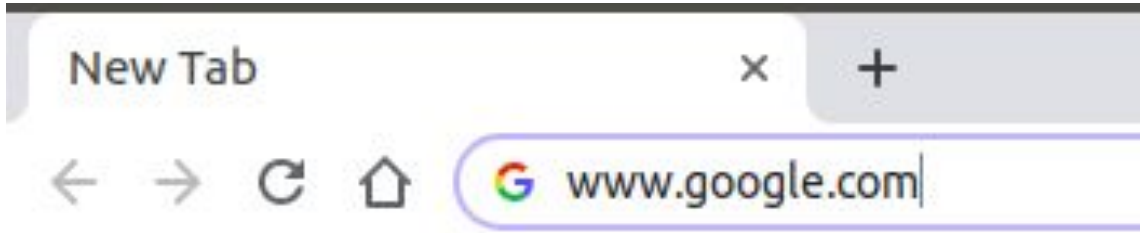
Components Involved

Resources: Code files such as HTML, CSS, JavaScript files and Assets such as images, videos, audios, pdfs ...

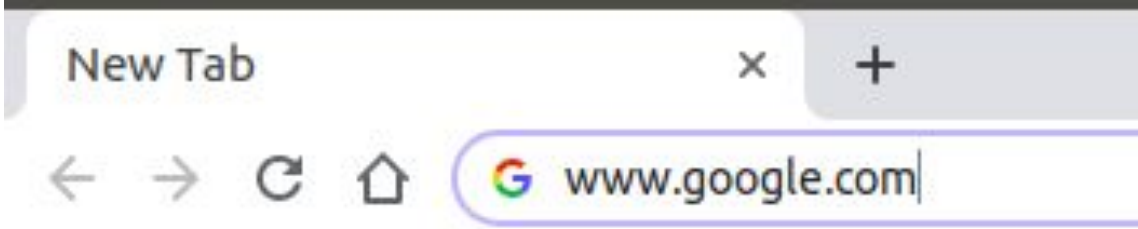
URL: Uniform Resource Locator which describes resources on the internet

How the web works

What will happen when you type a web address into your browser?



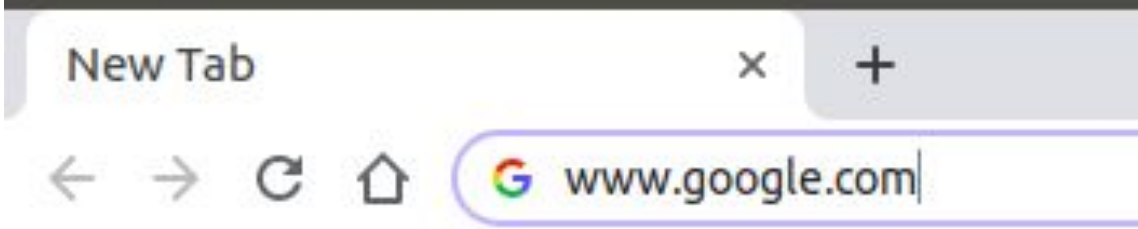
How the web works



What will happen when you type a web address into your browser

- The browser sends name resolution request to the DNS server, and finds the real address of the server that the website lives on (`www.google.com -> 216.58.210.68`)
- The **browser sends an HTTP request message** to the server, asking it to send a copy of the website to it. This message, and all other data sent between the client and the server, is sent across your internet connection using TCP/IP

How the web works

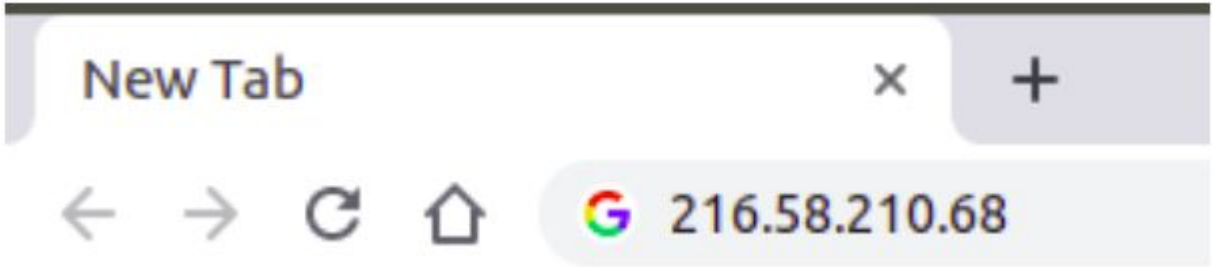


What will happen when you type a web address into your browser

- If the server approves the client's request, **the server sends the client a "200 OK" message** and then starts sending the website's files to the browser as a series of small chunks called data packets
- The **browser assembles the small chunks into a complete website** and displays it to you

How the web works

What will happen when you type a web address into your browser?



How the web works

Components of URL

`https://www.google.com/search?safe=active&oq=golang`

Format

`scheme://host[:port#]/path/.../[?query-string][#anchor]`

The parts in the **`square bracket`** are optional

Identify the `URL` components in the above example

How the web works

Components of URL

`scheme://host[:port#]/path/.../[?query-string][#anchor]`

`scheme`: specifies the protocol such as **`http`**, **`https`**, **`ftp`**, **`mailto`** ...

`host`: domain name or ip address of the web server

`port#`: indicates the technical "gate" used to access the resources on the web server

How the web works

Components of URL

`scheme://host[:port#]/path/.../[?query-string][#anchor]`

`path`: the path to the resource on the Web server

`?Query-string`**:** are extra parameters provided to the Web server. Those parameters are a **list of `key/value` pairs** separated with the `&` symbol

`#anchor`: an anchor to another part of the resource itself. An anchor represents a sort of "bookmark" inside the resource, giving the browser the directions to show the content located at that "bookmarked" spot

Basic HTTP Server in Go

```go
package main

import "net/http"

func index(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("<h1>Hello World!</h1>"))
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", index)
    http.ListenAndServe(":8080", mux)
}
```

`net/http` package

Package http **provides HTTP client and server** implementations

Check the `net/http` package documentation on the following link

https://golang.org/pkg/net/http/

http.NewServeMux function

```go
mux := http.NewServeMux()
```

Creates a new HTTP Request Multiplexer

HTTP Request Multiplexer: **matches** the **URL of incoming requests against a list of registered paths** and **calls the associated handler for the path** whenever a match is found

```go
9   func main() {
10      mux := http.NewServeMux()
11      mux.HandleFunc("/", index)
12      http.ListenAndServe(":8080", mux)
13  }
```

mux.HandleFunc function

Used for registering a given path (“/” in this case) with a handler function (**index**)

```
9   func main() {
10      mux := http.NewServeMux()
11      mux.HandleFunc("/", index)
12      http.ListenAndServe(":8080", mux)
13  }
```

Handler Function

The handler function (`index`) should have the following signature

```go
func(w http.ResponseWriter, r *http.Request)
```

```go
5   func index(w http.ResponseWriter, r *http.Request) {
6       w.Write([]byte("<h1>Hello World!</h1>"))
7   }
```

Handler Function

The handler function (`index`) should have the following signature

```
func(w http.ResponseWriter, r *http.Request)
```

The `w` parameter is where you write your `text/html` response to. It implements a `Write()` method which accepts a `slice` of `bytes`

The `r` parameter represents the HTTP request received from the client

```
5   func index(w http.ResponseWriter, r *http.Request) {
6       w.Write([]byte("<h1>Hello World!</h1>"))
7   }
```

`http.ListenAndServe` function

The `http.ListenAndServe` function is used to start the server at specific address and port

When it receives an HTTP request, it will hand it off to the **multiplexer** that we supply as the second argument (`mux`)

```go
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", index)
    http.ListenAndServe(":8080", mux)
}
```

Exercise

Write a Simple Web server that returns your name (use <h1> tag) and short description (use <p> tag) about your self when you send a http request to **/name** path

The `html/template` package

The `html/template` package is part of the Go standard library

We can use `html/template` to keep the HTML in a separate file

It allows us to change the html code without modifying the underlying Go code

Let us put the `<h1>Hello World!</h1>` html code in a separate `index.html` file

```
5   func index(w http.ResponseWriter, r *http.Request) {
6       w.Write([]byte("<h1>Hello World!</h1>"))
7   }
```

The `html/template` package

Add the following `html` code on `index.html` file

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Web Programming I</title>
    </head>
    <body>
        <h1>Hello World</h1>
    </body>
</html>
```

The html/template package

```go
3   import (
4       "html/template"
5       "net/http"
6   )
7
8   var templ = template.Must(template.ParseFiles("index.html"))
9
10  func index(w http.ResponseWriter, r *http.Request) {
11      templ.Execute(w, nil)
12  }
13
14  func main() {
15      mux := http.NewServeMux()
16      mux.HandleFunc("/", index)
17      http.ListenAndServe("localhost:8080", mux)
18  }
```

Parsing Templates from Files

In the `index` function, we execute the template created by providing two arguments to `templ.Execute` function:

> The first is where we want to write the output to (`w`), and the second one is the data we want to pass to the template (in this case `nil`)

```go
var templ = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    templ.Execute(w, nil)
}
```

Parsing Templates from Files

Template can either be parsed from a string or a file on disk.

The call to `template.ParseFiles` in the following code parses the `index.html` file in the root of the project directory and validates it

```
var templ = template.Must(template.ParseFiles("index.html"))
```

Exercise

Move your name and description information from the Go code to a separate .html file and update your code to use templates

Serving static files

Let us add `style.css` file inside `assets` subdirectory where the main .go file is stored in order to add some style to the `index.html` page

Though the stylesheet document is linked in the index.html page as shown in the next slide, it will not work

One way of making it work is to declare explicit handlers for the style.css file. However having a handler for all static files is not realistic, and cannot scale

Go provides a way to create one handler to take care of serving all static assets

Serving static files

`index.html` file

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>ITSE-3193</title>
  <link rel="stylesheet" href="/assets/style.css">
</head>
<body>
  <main>
    <header>
      Welcome to Web Programming I
    </header>
    <section class="container">
        <p>Lorem ipsum dolor sit amet magna aliqua</p>
    </section>
  </main>
</body>
</html>
```

Serving static files

`/assets/style.css` file

```
1   header {
2     width: 100%;
3     height: 50px;
4     background-color:   #E0f030;
5     padding: 10px;
6   }
7   .container {
8     width: 100%;
9     max-width: 720px;
10    margin: 0 auto;
11    padding: 10px;
12  }
```

Serving static files

Notice the added codes on line 17 and 18

```
8   var templ = template.Must(template.ParseFiles("index.html"))
9
10  func index(w http.ResponseWriter, r *http.Request) {
11      templ.Execute(w, nil)
12  }
13
14  func main() {
15
16      mux := http.NewServeMux()
17      fs := http.FileServer(http.Dir("assets"))
18      mux.Handle("/assets/", http.StripPrefix("/assets/", fs))
19      mux.HandleFunc("/", index)
20      http.ListenAndServe("127.0.0.1:8080", mux)
21  }
```

Serving static files

The `http.FileServer` function in the following code creates a handler that will serve files from a given directory (“assets”)

```
fs := http.FileServer(http.Dir("assets"))
```

The `http.StripPrefix` function removes the given prefix from the request URL ’s path

Exercise

Style your page with some CSS

Passing Data to Templates

Suppose we have the following Course struct and its instance that we want to pass to the template

```go
type Course struct {
    Title       string
    ECTS        int
    Code        string
    Description string
}
```

```go
course := Course{"DLD", 7, "ITSE-3182", "Lorem Ipsum"}
```

Passing Data to Templates

Notice the change on line 18

```
9   type Course struct {
10      Title, Code, Description string
11      ECTS                     int
12  }
13
14  var templ = template.Must(template.ParseFiles("index.html"))
15
16  func index(w http.ResponseWriter, r *http.Request) {
17      course := Course{"DLD", "ITSE-3182", "Lorem Ipsum", 7}
18      templ.Execute(w, course)
19  }
```

Passing Data to Templates

We can use the data passed to our template on our `index.html` page

```
7   <body>
8     <main>
9       <header>
10        Welcome to {{ .Title }}
11      </header>
12      <section class="container">
13          <ul>
14              <li>ECTS: {{ .ECTS }}</li>
15              <li>Code: {{ .Code }}</li>
16          </ul>
17          <p>{{ .Description }}</p>
18      </section>
19    </main>
20  </body>
```

Exercise

Put your information in a struct called Student and create an instance of it and display the information on your page

Questions

Explain the purpose of the following components

- `http` and `template` packages
- `HandleFunc`
- `FileServer`
- `ListenAndServe`
- Multiplexer

Next

Read about

- HTTP Versions (0.9, 1.x, HTTP/2, HTTP/3)
- SPDY, QUIC, TLS 1.3
- HTTP Request and Response Headers
- HTTP Request Methods and Response Status