# WEB programming

## HTTP server

The Go standard library provides built-in support for creating an HTTP server to serve your web content or making HTTP requests to those servers. Writing a simple HTTP server is a simple task in GO language. Here is an example published on [DigitalOcean](https://www.digitalocean.com/community/tutorials/how-to-make-an-http-server-in-go) site:

package main

import (

"fmt"

"io"

"log"

"net/http"

)

func getRoot(w http.ResponseWriter, r \*http.Request) {

fmt.Printf("got / request\n")

io.WriteString(w, "This is my website!\n")

}

func getHello(w http.ResponseWriter, r \*http.Request) {

fmt.Printf("got /hello request\n")

io.WriteString(w, "Hello HTTP!\n")

}

func main() {

mux := **http.NewServeMux()**

mux.HandleFunc("/", getRoot)

mux.HandleFunc("/hello", getHello)

log.Fatal(**http.ListenAndServe(":8080", mux)**)

}

#### ListenAndServe

The main work in supplied program is done by the function [ListenAndServe](https://pkg.go.dev/net/http" \l "ListenAndServe).

func ListenAndServe(addr string, handler Handler) error

The http package documentation states that the second parameter of **ListenAndServe** function may be **nil**. In this case, we connect the router functions directly to the **http** object:

http.HandleFunc("/", getRoot)

http.HandleFunc("/hello", getHello)

log.Fatal(http.ListenAndServe(":8080", nil))

The **http.ListenAndServeTLS** function is designed for the HTTPS protocol and performs the same functions as http.ListenAndServe: listens on the TCP network address **addr** and then calls Serve with **handler** to handle requests on incoming connections:

func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error

Parameters:

* **addr** - the TCP network address should be in the format "**host:port**". If you omit the **host** (like we did with ":8080") then the server will listen on all your computer’s available network interfaces. You might sometimes see network addresses written using named ports like "**:http**" or "**:http-alt**" instead of a number. If you use a named port then Go will attempt to look up the relevant port number from your **/etc/services** file when starting the server, or will return an error if a match can’t be found. A slash at the end of the address indicates that it is the top of the tree. Longer patterns take precedence over shorter ones, so that if there are handlers registered for both "**/images/**" and "**/images/thumbnails/**", the latter handler will be called for paths beginning "**/images/thumbnails/**" and the former will receive requests for any other paths in the "**/images/**" subtree.
* **handler** - Go's **net/http** package ships with the simple but effective http.ServeMux [handler](https://pkg.go.dev/net/http" \l "ServeMux), plus a few functions to generate common handlers including **http.FileServer()**, **http.NotFoundHandler()** and **http.RedirectHandler()**. **Handler** is interface thus anything that satisfies the [http.Handler](https://pkg.go.dev/net/http" \l "Handler) interface, may be placed in the last parameter. An example of custom handler is presented on [alexedwards](https://www.alexedwards.net/blog/an-introduction-to-handlers-and-servemuxes-in-go) site, see section "Custom handlers".
* **certFile** - path to the SSL certificate file,
* **keyFile** - path to the private key file.

#### HTTPS protocol

* + - 1. When working with the HTTPS protocol, the .NET CLI creates a new certificate for each new project. After a while, when you dive into Trusted Root Certificated Authorities, you will see a bunch of certificates that you no longer have or don't remember for which project you created them. [Lewel Murithi](https://www.section.io/engineering-education/how-to-get-ssl-https-for-localhost/) proposed an elegant solution to this problem. The idea is this: you create a long-term self-signed certificate and place it into Trusted Root Certificated Authorities. Certificates for new projects will be signed with this certificate of your own CA. Obviously, this solution is only suitable for local networks, but Trusted Root Certificated Authorities will not be spammed.
      2. In order to start a server working under the HTTPS protocol, you need to generate an SSL certificate and a public-private key pair. The [OpenSSL](https://www.openssl.org/) command line interface (CLI) and toolkit allows you to generate SSL certificates, private keys, Certificate signing requests (CSR) and perform other kinds of cryptography operations. You need to create private key **.key** file and a certificate. Follow this instruction:

1. Generate **encrypted private key**

mkdir cert

cd cert

mkdir CA // Your own „Certificate authority“

cd CA

openssl genrsa -out CA.key -des3 3072

These commands will ask you a password. Select simple one and confirm it on next question.

1. Linux browsers require Subject Alternative Names. They can only be created via a configuration file. Create the text file given below and name it CA.CNF:

[req]

distinguished\_name = req\_distinguished\_name

req\_extensions = req\_ext

prompt = no

[req\_distinguished\_name]

C = LT

ST = N\_A

L = Kaunas

O = Home Computer

OU = IT

CN = localhost

[req\_ext]

subjectAltName = @alt\_names

[alt\_names]

IP.1 = 127.0.0.1

IP.2 = 192.168.1.107

IP.3 = 192.168.1.171

DNS.1 = localhost

DNS.2 = KIBIRVIBIR-PC

DNS.3 = gediminas-TravelMate-5760Z

Parameters:

**C** - country (2 letter code),

**ST** – state or Province name (full name),

**L** - Locality Name (e.g., city),

**O** - Organization Name (e.g., company),

**OU** - Organizational Unit Name (e.g., section)

**CN** - The fully-qualified domain name (FQDN) (e.g., [www.example.com](http://www.example.com/)),

**IP.x** – IP addresses, only firs row is mandatory, IP.2, IP.3 are address in my local network,

**DNS.x** - subject alternative names, once again only first name is mandatory, other two ones are names in my local network.

1. Generate a **root CA certificate** using the key generated, that will be valid for ten years

openssl req -x509 -sha256 -new -nodes -days 3650 -key CA.key \

-out CA.pem -config CA.CNF

A certificate for Windows can be created without a configuration file:

openssl req -x509 -sha256 -new -nodes -days 3650 -key CA.key -out CA.pem \

-subj "/C=LT/ST=N\_A/L=Kaunas/O=Home Computer/OU=IT/CN=localhost"

The values specified in the **subj** parameter (/C, /ST, /L, /O, /OU, /CN) are the same as in the configuration file.

Certificate importing tool on Linux requires CRT extension, thus copy the key:

cp CA.pem CA.crt

Certificate importing tool requires PFX format. Convert pem into pfx with command:

openssl pkcs12 -export -out CA.pfx -inkey CA.key -in CA.pem

Windows users must import CA.pfx file into **Trusted Root Certification Authorities**. Import root certificate with mmc snap-in. Run mmc, select File → Add/Remove Snap-in → Certificates → My user account → OK. Open the new snap-in and select Trusted Root Certification Authorities. Right click on Certificates → All Tasks → Import… Import the CA.pfx file from CA directory. You will find the details on  [thewindowsclub](https://www.thewindowsclub.com/manage-trusted-root-certificates-windows?utm_content=cmp-true) site.

Linux users must run the following commands in a terminal:

sudo apt-get install -y ca-certificates

sudo cp CA.crt /usr/local/share/ca-certificates

sudo update-ca-certificates

After this correction Ubuntu tools (curl, wget, …) will accept your certificates but **Chrome** and **Mozilla** browsers require you to upload the root certificate to their own repositories. Open site **about:preferences#privacy** in Mozilla browser and click the button "**View certificates…**". Click on button "**Import…**" and import the CA.crt file.

Open site **chrome://settings/certificates** in Chrome browser and click on "**Import**" button. Select the **CA.pfx** file. Open site **chrome://flags/#allow-insecure-localhost** after that. Enable button "**Allow invalid certificates for resources loaded from localhost.**".

1. Being in CA directory create **localhost** subfolder with **localhost.ext** inside:

mkdir localhost // New project

cd localhost

touch localhost.ext

Windows users have no the **touch** command. Use command **type** in this case:

type nul > localhost.ext

1. Write this information into localhost.ext:

authorityKeyIdentifier = keyid,issuer

basicConstraints = CA:FALSE

keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment

subjectAltName = @alt\_names

[alt\_names]

IP.1 = 127.0.0.1

IP.1 = 127.0.0.1

IP.2 = 192.168.1.107

DNS.1 = localhost

DNS.2 = KIBIRVIBIR-PC

DNS.3 = gediminas-TravelMate-5760Z

Parameters **localhost** and **127.0.0.1** are mandatory, other adresses are optional. An example contains settings from my home network. One can decide to add more domains or IP addresses to the file but ensure to edit the **/etc/hosts** file to have those domains point to the local machine (127.0.0.1). Windows users will find the **hosts** file in C:\**Windows\System32\drivers\etc** directory.

1. Generate private key with command:

openssl genrsa -out localhost.key -des3 3072

1. Generate CSR (signing request file) using the key:

openssl req -new -key localhost.key -out localhost.csr -config ../CA.CNF

Windows users can generate the CSR file without config:

openssl req -new -key localhost.key -out localhost.csr \

-subj "/C=LT/ST=N\_A/L=Kaunas/O=Home Computer/OU=IT/CN=localhost"

1. With this CSR, we can request the CA to sign a certificate as below. Since we are creating the certificate on the localhost computer, we will use our own Certificated Authority (CA.\* files). Note that the paths for **CA.key** and **CA.pem** files are dependent on where the user is running commands from. In this case, the commands below are ran from **/cert/CA/localhost**:

openssl x509 -req -in localhost.csr -CA ../CA.pem -CAkey ../CA.key \

-CAcreateserial -days 3650 -sha256 -extfile localhost.ext -out localhost.crt

1. The server will need the localhost.crt certificate file, and the decrypted key since our localhost.key is in encrypted form. Decrypt the localhost.key and store that file:

openssl rsa -in localhost.key -out localhost.decrypted.key

1. Windows users can import root certificate with **mmc** snap-in. Run mmc, select **File → Add/Remove Snap-in → Certificates → My user account → OK.** Open the new snap-in and select **Trusted Root Certification Authorities**. Right click on  **Certificates → All Tasks → Import…** Import the CA.pfx file from CA directory. Launch WEB server with ListenAndServeTLS function. Any browser would accept [https://localhost:8080](https://localhost:8080/) address. Sometimes this URL is occupied by site registered on your local IIS. Stop your local IIS and repeat the test. Chrome browser will inform you that the certificate is insecure but the site will be displayed.

#### Serving Static Files and Sites

The net/http package has a **FileServer** function that allows you to display static pages:

package main

import (

"log"

"net/http"

)

func main() {

fs := http.**FileServer**(http.Dir("./static"))

http.Handle("/", fs)

log.Print("Listening on :3000...")

err := http.ListenAndServe(":3000", nil)

if err != nil {

log.Fatal(err)

}

}

The provided file server will show any file from the ./static directory. Just type the command

localhost:3000/file\_name

in a browser’s address window. Put this an **example.html** file into the **./static** directory for testing the application

<!doctype html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1">

<title>A static page</title>

<link rel="stylesheet" href="styles/main.css">

</head>

<body>

<h1>Hello from static page</h1>

</body>

</html>

The page requires stylesheet in the styles subdirectory. Use this one:

body {

color: #c0392b

}

Structure of the project:

**home**

go.mod

main.go

**static**

example.html

**styles**

main.css

Name of the directories are typed in bold. The browser’s command localhost:3000/example.html will show you text „Hello from static page“. Alex Edwards explained this [example](https://www.alexedwards.net/blog/serving-static-sites-with-go).

**http.FileServer** is also used in dynamic sites. It serves CSS files, images, and other objects that can be downloaded by the client. For that purpose, you have to use the request multiplexer:

func main() {

fs := http.FileServer(http.Dir("public"))

http.Handle("/public/", http.StripPrefix("/public/", fs))

http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {

...

})

http.HandleFunc("/abc", func(w http.ResponseWriter, r \*http.Request) {

...

})

...

http.ListenAndServe(":8080", nil)

}

In this example, the static files are placed in the "public" directory. The file server will handle all requests starting with "/public/" :

<link rel="stylesheet" href="public/css/format.css">

The file server's root directory can have its own subdirectories, and its name can differ from the pattern specified in the http.Handle function. A complete example of such WEB application is provided at the [zetcode](https://zetcode.com/golang/http-serve-static-files/) site

#### Text templates

Any Go program can use the [text/template](https://pkg.go.dev/text/template) or [html/template](https://pkg.go.dev/html/template) package—both included in the Go standard library. Both packages allow you to write textual templates and pass data into them to render a document formatted to your liking. Package template (**html/template**) implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as **text/template** and should be used instead of text/template whenever the output is HTML. Within the templates, you can loop over the data and use conditional logic to decide which items to include in the document and how they should appear. Templates live either as strings in your code or in their own files alongside your code. They contain boilerplate static text interlaced with conditional statements (i.e. if/else), flow control statements (i.e. loops), and function calls, all wrapped within {{. . .}} markers. You will pass some data into your template to render a final document.

One must follow this instruction working with templates:

1. import package **text/template** or **html/template,**
2. create a template using the operator **template.New(template\_name)**
3. compile the template using the **Parse()** function, this function has several modifications (**ParseFS**, **ParseFiles**, **ParseGlob**),
4. execute the compiled template using the **Exec()** function.

A complete description of the tex/template package is available at the [GO](https://pkg.go.dev/text/template" \l "pkg-overview) site. Here is an example of this scenario:

// file main.go

// https://www.digitalocean.com/community/tutorials/how-to-use-templates-in-go

package main

import (

"os"

"text/template"

)

type **Pet** struct {

Name string

Sex string

Intact bool

Age string

Breed string

}

func main() {

**dogs** := []Pet{

{

Name: "Jutube",

Sex: "Female",

Intact: false,

Age: "10 months",

Breed: "German Sheperd/Pitbul",

},

{

Name: "Zephyr",

Sex: "Male",

Intact: true,

Age: "13 years, 3 months",

Breed: "German Sheperd/Border Collie",

},

{

Name: "Bruce Wayne",

Sex: "Male",

Intact: false,

Age: "3 years, 8 months",

Breed: "Chihuahua",

},

}

var tmplFile = "pets.tmpl"

tmpl, err := template.**New**(tmplFile).**ParseFiles**(tmplFile)

if err != nil {

panic(err)

}

err = tmpl.**Execute**(os.Stdout, dogs)

if err != nil {

panic(err)

}

}

The template may be written as string contant. It is in a separate file here:

{{/\* file pets.tmpl \*/}}

Number of dogs: {{ . | len -}}

{{ range . }}

---

Name: {{ .Name }}

Sex: {{ .Sex }} ({{ if .Intact }}intact{{ else }}{{ if (.Sex | eq "Female") }}spayed{{ else }}neutered{{end}}{{ end }})

Age: {{ .Age }}

Breed: {{ .Breed }}

{{ end }}

The input text for a template is UTF-8-encoded text in any format. "**Actions**"--data evaluations or control structures--are delimited by **"{{"** and **"}}"**; all text outside actions is copied to the output unchanged. Write "- " (minus with space) immediately after the opening delimiter or " -" (space with minus) before the closing delimiter if you want to trim spaces before or after the action. **Actions** represent the data evaluations, functions or control loops. They access the data via the cursor, denoted by a dot (.). Initial value for the cursor supplies second parameter in the **Execute** command:

err = tmpl.**Execute**(os.Stdout, dogs)

Slice **dogs** is cursor for the template. The data passed into a template may be absolutely anything, but it is common to pass in a slice, array, or map—something iterable.

Arguments

An argument is a simple value, denoted by one of the following:

* A **boolean**, **string**, **character**, **integer**, **floating-point**, **imaginary** or **complex constant** in Go syntax. These behave like Go's untyped constants.
* The keyword **nil**, representing an untyped Go nil.
* The character '.' (period): . The result is the current value of the dot.
* A variable name, which is a (possibly empty) alphanumeric string preceded by a dollar sign, such as $piOver2 or $ The result is the value of the variable.
* The name of a field of the data, which must be a **struct**, preceded by a period, such as **.Field**; the result is the value of the field. Field invocations may be chained: .Field1.Field2 Fields can also be evaluated on variables, including chaining: $x.Field1.Field2.
* The name of a key of the data, which must be a map, preceded by a period, such as .Key The result is the map element value indexed by the key. Key invocations may be chained and combined with fields to any depth: .Field1.Key1.Field2.Key2 Although the key must be an alphanumeric identifier, unlike with field names they do not need to start with an upper case letter. Keys can also be evaluated on variables, including chaining: **$x.key1.key2**.
* The name of a niladic method of the data, preceded by a period, such as .Method The result is the value of invoking the method with dot as the receiver, dot.Method(). Such a method must have one return value (of any type) or two return values, the second of which is an error. If it has two and the returned error is non-nil, execution terminates and an error is returned to the caller as the value of Execute. Method invocations may be chained and combined with fields and keys to any depth: .Field1.Key1.Method1.Field2.Key2.Method2 Methods can also be evaluated on variables, including chaining: $**x.Method1.Field**.
* The name of a niladic function defined inside the template, such as fun The result is the value of invoking the function, fun(). The return types and values behave as in methods.
* A parenthesized instance of one the above, for grouping.

Arguments may evaluate to any type; if they are pointers the implementation automatically indirects to the base type when required. If an evaluation yields a function value, such as a function-valued field of a struct, the function is not invoked automatically, but it can be used as a truth value for an if action and the like.

Pipeline

A pipeline is a possibly chained sequence of "commands". A command is a simple value (argument) or a function or method call, possibly with multiple arguments:

**Argument**

The result is the value of evaluating the argument.

**.Method [Argument...]**

The method can be alone or the last element of a chain but,

unlike methods in the middle of a chain, it can take arguments.

The result is the value of calling the method with the

arguments:

dot.Method(Argument1, etc.)

**functionName [Argument...]**

The result is the value of calling the function associated

with the name:

function(Argument1, etc.)

A pipeline may be "chained" by separating a sequence of commands with pipeline characters '|'. In a chained pipeline, the result of each command is passed as the last argument of the following command. The output of the final command in the pipeline is the value of the pipeline. The output of a command will be either one value or two values, the second of which has type error. If that second value is present and evaluates to non-nil, execution terminates and the error is returned to the **caller** of Execute.

Variable

A pipeline inside an action may initialize a variable to capture the result:

$variable := pipeline

where $variable is the name of the variable. An action that declares a variable produces no output. Variables previously declared can also be assigned, using the syntax

$variable = pipeline

If a "range" action initializes a variable, the variable is set to the successive elements of the iteration. Also, a "range" may declare two variables, separated by a comma:

range $element := pipeline

range $index, $element := pipeline

A variable's scope extends to the "end" action of the control structure ("if", "with", or "range") in which it is declared, or to the end of the template if there is no such control structure. A template invocation does not inherit variables from the point of its invocation.

When execution begins, $ is set to the data argument passed to Execute, that is, to the starting value of dot.

Comments

The parser ignores comments. The comment may contain newlines. Comments do not nest and must start and end at the delimiters:

{{/\* a comment \*/}}

{{- /\* a comment with white space trimmed from preceding and following text \*/ -}}

Values

{{pipeline}}

The default textual representation (the same as would be printed by fmt.Print) of the value of the pipeline is copied to the output.

Conditionals

{{if \_cond\_pipeline\_}} T1 {{end}}

If the value of the pipeline is empty, no output is generated; otherwise, T1 is executed. The empty values are false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero. Dot is unaffected.

{{if \_cond\_pipeline\_}} T1 {{else}} T0 {{end}}

If the value of the pipeline is empty, T0 is executed; otherwise, T1 is executed. Dot is unaffected.

{{if \_cond\_pipeline\_}} T1 {{else if \_cond\_pipeline\_}} T0 {{end}}

This note is in abbreviated form. It means exactly the same as this action:

**{{if \_cond\_pipeline\_}} T1 {{else}}{{if \_cond\_pipeline\_}} T0 {{end}}{{end}}**

The **\_cond\_pipeline\_** can be a **boolean field** or a **function** that produces a boolean result. The parser has the following boolean functions:

**eq** Returns the boolean truth of arg1 == arg2

**ne** Returns the boolean truth of arg1 != arg2

**lt** Returns the boolean truth of arg1 < arg2

**le** Returns the boolean truth of arg1 <= arg2

**gt** Returns the boolean truth of arg1 > arg2

**ge** Returns the boolean truth of arg1 >= arg2

**and** Returns the boolean AND of its arguments

**or** Returns the boolean OR of its arguments

**not** Returns the boolean negation of its single argument

Loops

{{range pipeline}} T1 {{end}}

The value of the pipeline must be an array, slice, map, or channel. If the value of the pipeline has length zero, nothing is output; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed. If the value is a map and the keys are of basic type with a defined order, the elements will be visited in sorted key order.

{{range pipeline}} T1 {{else}} T0 {{end}}

The value of the pipeline must be an array, slice, map, or channel. If the value of the pipeline has length zero, dot is unaffected and T0 is executed; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed.

{{break}}

The innermost {{range pipeline}} loop is ended early, stopping the current iteration and bypassing all remaining iterations.

{{continue}}

The current iteration of the innermost {{range pipeline}} loop is stopped, and the loop starts the next iteration.

A name of the template

The template can be named using:

1. Function **[template.New](https://pkg.go.dev/text/template" \l "New)**. The function allocates a new, undefined template with the given name.

template.New(name string) \*Template

1. Function [**template.ParseFiles**](https://cs.opensource.google/go/go/+/go1.21.1:src/text/template/helper.go;l=40). ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the base name and parsed contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned \*Template is nil.

template.ParseFiles(filenames ...string) (\*Template, error)

1. Writing action **define** inside the template:

{{**define** "footer"}} // Name of the template is „footer“

<footer>

<p>

Copyright 2016 Calhoun.io

</p>

<p>

Contact information: <a href="mailto:jon@calhoun.io">jon@calhoun.io</a>.

</p>

</footer>

{{**end**}}

Execute template

{{template "name"}}

The template with the specified name is executed with nil data.

{{template "name" pipeline}}

The template with the specified name is executed with dot set to the value of the pipeline.

Define and execute template

{{block "name" pipeline}} T1 {{end}}

A block is shorthand for defining a template

**{{define "name"}} T1 {{end}}**

and then executing it in place

**{{template "name" pipeline}}**

Change a value of the dot

{{with pipeline}} T1 {{else}} T0 {{end}}

If the value of the pipeline is empty, dot is unaffected and T0 is executed; otherwise, dot is set to the value of the pipeline and T1 is executed.

Functions

The template engine has a wide list of embedded functions. This list is published on [GO](https://pkg.go.dev/text/template" \l "hdr-Functions) site. During execution functions are found in two function maps: first in the template, then in the global function map. By default, no functions are defined in the template but the Funcs method can be used to add them.

The template can call the function described in a context:

// file main.go

type ViewData struct {

User User

}

type User struct {

ID int

Email string

}

func (u User) HasPermission(feature string) bool {

if feature == "feature-a" {

return true

} else {

return false

}

}

...

// file hello.html; Context is ViewData

{{if **.User.HasPermission** "feature-a"}}

<div class="feature">

<h3>Feature A</h3>

<p>Some other stuff here...</p>

</div>

{{else}}

<div class="feature disabled">

<h3>Feature A</h3>

<p>To enable Feature A please upgrade your plan</p>

</div>

{{end}}

It is necessary to use action **call** when context has property of a **func** type:

{{if (call .User.HasPermission "feature-a")}}

<div class="feature">

<h3>Feature A</h3>

<p>Some other stuff here...</p>

</div>

{{else}}

<div class="feature disabled">

<h3>Feature A</h3>

<p>To enable Feature A please upgrade your plan</p>

</div>

{{end}}

Custom functions can be placed in a dictionary of type **map[string]any,** see description of [FuncMap](https://pkg.go.dev/text/template" \l "FuncMap) function:

func main() {

var err error

testTemplate, err = template.New("hello.gohtml").**Funcs**(**template.FuncMap**{

"**hasPermission**": func(user User, feature string) bool {

return user.ID == 1 && feature == "feature-a"

},

}).ParseFiles("hello.gohtml")

if err != nil {

panic(err)

}

http.HandleFunc("/", handler)

http.ListenAndServe(":3000", nil)

}

You don’t need the call action in this case:

{{if hasPermission .User "feature-a"}}

...

{{if hasPermission .User "feature-b"}}

...

Template functions are described on a [calhoun](https://www.calhoun.io/intro-to-templates-p3-functions/) site.

### Gorilla mux

Package gorilla/mux implements a request router and dispatcher for matching incoming requests to their respective handler.

* 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.

Install the package with **go get**:

go get -u github.com/gorilla/mux

You can use GorillaMux exactly like http.ServeMux:

func main() {

r := mux.NewRouter()

r.HandleFunc("/", HomeHandler)

r.HandleFunc("/products", ProductsHandler)

r.HandleFunc("/articles", ArticlesHandler)

http.ListenAndServe(":3000", r)

}

Here we register three routes mapping URL paths to handlers. This is equivalent to how http.HandleFunc() works: if an incoming request URL matches one of the paths, the corresponding handler is called passing (http.ResponseWriter, \*http.Request) as parameters.

Paths can have variables. They are defined using the format {name} or {name:pattern}. If a regular expression pattern is not defined, the matched variable will be anything until the next slash. For example:

r := mux.NewRouter()

r.HandleFunc("/products/{key}", ProductHandler)

r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)

r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)

The names are used to create a map of route variables which can be retrieved calling mux.Vars():

func ArticlesCategoryHandler(w http.ResponseWriter, r \*http.Request) {

vars := mux.Vars(r)

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Category: %v\n", vars["category"])

}

Routes can be restricted using predefined functions:

* **Domain or subdomain**

r := mux.NewRouter()

r.Host("[www.example.com](http://www.example.com/)") // Matches if domain is "www.example.com".

r.Host("{subdomain:[a-z]+}.example.com") // Matches a dynamic subdomain.

* **Path prefix** of the requests

r.PathPrefix("/products/")

* **HTTP methods**

r.Methods("GET", "POST")

* **URL schemes**

r.Schemes("https")

* **Header values**

r.Headers("X-Requested-With", "XMLHttpRequest")

Regex support also exists for matching Headers within a route:

r.HeadersRegexp("Content-Type", "application/(text|json)")

This route will match both requests with a **Content-Type** of **application/json** as well as **application/text**.

* **Query values**

r.Queries("key", "value")

* **Custom matcher function** which returns true or false

r.MatcherFunc(func(r \*http.Request, rm \*RouteMatch) bool {

return r.ProtoMajor == 0

})

It is possible to combine several matchers in a single route:

r.HandleFunc("/products", ProductsHandler).

Host("www.example.com").

Methods("GET").

Schemes("http")

Routes are tested in the order they were added to the router. If two routes match, the first one wins:

r := mux.NewRouter()

r.HandleFunc("/specific", specificHandler)

r.PathPrefix("/").Handler(catchAllHandler)

The **Subrouter()** function allows you to group routing rules. Selection conditions may be specified for this function. These conditions will be applied to all members of the group:

r := mux.NewRouter()

s := r.Host("www.example.com").Subrouter()

s.HandleFunc("/products/", ProductsHandler)

s.HandleFunc("/products/{key}", ProductHandler)

s.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)

The three URL paths we registered above will only be tested if the domain is [**www.example.com**](http://www.example.com/). When a subrouter has a path prefix, the inner routes use it as base for their paths:

r := mux.NewRouter()

s := r.PathPrefix("/products").Subrouter()

s.HandleFunc("/", ProductsHandler) // "/products/"

s.HandleFunc("/{key}/", ProductHandler) // "/products/{key}/"

// "/products/{key}/details"

s.HandleFunc("/{key}/details", ProductDetailsHandler)

Host and path may be defined separately use subrouters:

r := mux.NewRouter()

s := r.Host("{subdomain}.example.com").Subrouter()

s.Path("/articles/{category}/{id:[0-9]+}").

HandlerFunc(ArticleHandler).

Name("article")

Path provided to **PathPrefix()** represents a "wildcard": calling **PathPrefix("/static/").Handler(...)** means that the handler will be passed any request that matches "**/static/\***". This makes it easy to serve static files with mux:

func main() {

var dir string

flag.StringVar(&dir, "dir", ".", "the directory to serve files from. Defaults to the current dir")

flag.Parse()

r := mux.NewRouter()

// This will serve files under http://localhost:8000/static/<filename>

r.PathPrefix("/static/").Handler(http.StripPrefix("/static/", http.FileServer(http.Dir(dir))))

srv := &http.Server{

Handler: r,

Addr: "127.0.0.1:8000",

// Good practice: enforce timeouts for servers you create!

WriteTimeout: 15 \* time.Second,

ReadTimeout: 15 \* time.Second,

}

log.Fatal(srv.ListenAndServe())

}

#### Named routes and redirection

Routes can be named. All routes that define a name can have their URLs built, or "**reversed**". We define a name calling **Name()** on a route:

r := mux.NewRouter()

r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler).

**Name("article")**

To build a URL, get the route and call the URL() method, passing a sequence of key/value pairs for the route variables:

url, err := r.**Get("article")**.URL("category", "technology", "id", "42")

This request result will produce a **url.URL** with the following path:

"/articles/technology/42"

This also works for host and query value variables:

r := mux.NewRouter()

r.Host("{subdomain}.domain.com").

Path("/articles/{category}/{id:[0-9]+}").

Queries("filter", "{filter}").

HandlerFunc(ArticleHandler).

Name("article")

Request

url, err := r.Get("article").URL("subdomain", "news",

"category", "technology",

"id", "42", "filter", "gorilla")

will return **url.URL** with the following path:

"http://news.domain.com/articles/technology/42?filter=gorilla"

All variables defined in the route are required, and their values must conform to the corresponding patterns. These requirements guarantee that a generated URL will always match a registered route.

There's also a way to build only the URL host or path for a route: use the methods **URLHost()** or **URLPath()** instead of **URL()**:

r := mux.NewRouter()

r.Host("{subdomain}.example.com").

Path("/articles/{category}/{id:[0-9]+}").

Queries("filter", "{filter}").

HandlerFunc(ArticleHandler).

Name("article")

...

// "http://news.example.com/"

host, err := r.Get("article").URLHost("subdomain", "news")

// "/articles/technology/42"

path, err := r.Get("article").URLPath("category", "technology", "id", "42")

To find all the required variables for a given route when calling URL(), the method GetVarNames() is available:

r := mux.NewRouter()

r.Host("{domain}").

Path("/{group}/{item\_id}").

Queries("some\_data1", "{some\_data1}").

Queries("some\_data2", "{some\_data2}").

Name("article")

// Will print [domain group item\_id some\_data1 some\_data2] <nil>

fmt.Println(r.Get("article").GetVarNames())

The **Walk** function on **mux.Router** can be used to visit all of the routes that are registered on a router. An example of this function you will find on site [https://pkg.go.dev/github.com/gorilla/mux#section-readme](https://pkg.go.dev/github.com/gorilla/mux" \l "section-readme).

### Graceful shutdown http server

The document "03\_GoLanguage.docx" shows how to stop the WEB server, see to the "Signals" section. Here we will show you a slightly different way that is compatible with the Gorilla.mux package:

package main

import (

"context"

"log"

"net/http"

"os"

"os/signal"

"syscall"

"time"

"github.com/gorilla/mux" // need to use dep for package management

)

func TestEndpoint(w http.ResponseWriter, r \*http.Request) {

w.WriteHeader(200)

w.Write([]byte("Test is what we usually do"))

}

func main() {

router := mux.NewRouter()

router.HandleFunc("/test", TestEndpoint).Methods("GET")

srv := &http.Server{

Addr: ":8080",

Handler: router,

}

done := make(chan os.Signal, 1)

signal.Notify(done, os.Interrupt, syscall.SIGINT, syscall.SIGTERM)

go func() {

if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {

log.Fatalf("listen: %s\n", err)

}

}()

log.Print("Server Started")

<-done

log.Print("Server Stopped")

ctx, cancel := context.WithTimeout(context.Background(), 5\*time.Second)

defer func() {

// extra handling here

cancel()

}()

if err := srv.Shutdown(ctx); err != nil {

log.Fatalf("Server Shutdown Failed:%+v", err)

}

log.Print("Server Exited Properly")

}

**srv.ListenAndServe()** may be launched on main thread but this function is blocking and capturing of the <-done channel must be performed inside endless go-routine. Document **03\_GoLanguage.docx** demonstrates how to capture a value from <-done chanel when **srv.ListenAndServe()** is on main thread. Pushing **srv.ListenAndServe()** into the go-routine simplifies handling of the channel.

**Serve** accepts incoming **HTTP** connections on the listener l, creating a **new service goroutine** for each. The service goroutines read requests and then call handler to reply to them.

if err := srv.Shutdown(ctx); err != nil {

log.Fatalf("Server Shutdown Failed:%+v", err)

}

**Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down. If the provided context expires before the shutdown is complete, Shutdown returns the context’s error, otherwise it returns any error returned from closing the Server’s underlying Listener(s).**

The quote above came from Go documentation said “and then waiting indefinitely for connections to return to idle and then shut down”, this part you should worry because it means some zombie connections may still be there and use your memory, in order to solve this, the safest way is to setup a timeout threshold using the Context:

ctx, cancel := context.WithTimeout(context.Background(), 5\*time.Second)

...

if err := srv.Shutdown(ctx); err != nil {

...

}

Large system will have http server, usually combined with multiple other connections like database, message queues, etc. And we need to handle these properly as well by defining the defer func and using cancel() to release these resources:

defer func() {

// Close database, redis, truncate message queues, etc.

cancel()

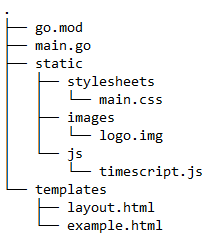
}()

// [Graceful shutdown in Go http server | by Sam Wang | honestbee-tw-engineering | Medium](https://medium.com/honestbee-tw-engineer/gracefully-shutdown-in-go-http-server-5f5e6b83da5a)

// [mux package - github.com/gorilla/mux - Go Packages](https://pkg.go.dev/github.com/gorilla/mux" \l "section-readme)

### Serving a complete web application

Consider this structure of the WEB application:



Here we have 2 problems:

1. specify the correct address for loading css, js and image files,
2. pass the parameter to the template parser.

#### With Http package

Problems are solved quite simply by working with a standard router (with http package, without ServeMux or GorillaMux):

1. Requests starting with the word "static" are directed to the file server:

fs := http.FileServer(http.Dir("./static"))

http.Handle("/static/", http.StripPrefix("/static/", fs))

After providing these settings, a link to the CSS file will look like this:

<link rel="stylesheet" type="text/css" href="**/static/stylesheets/main.css**">

In the first parameter of the http.Handle() function, you specify a name of the route. The parameter in the http.FileServer() function defines the name of the directory., These names match in the given example, but they can be different.

1. The second parameter of the **http.Handle** function is the **http.Handler** interface. Any object that implements this interface can be passed there. **http.Handler** interface has only one function - **ServeHTTP**. The following example demonstrates passing parameters with object:

type LayoutStruct struct {

StaticDir string

ListenPort string

Subject string

}

func (cnt LayoutStruct) ServeHTTP(w http.ResponseWriter, r \*http.Request) {

lp := filepath.Join("templates", "layout.html")

fp := filepath.Join("templates", filepath.Clean(r.URL.Path))

tmpl, err := template.ParseFiles(lp, fp)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

err = tmpl.ExecuteTemplate(w, "layout", cnt)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

}

func main() {

port := flag.String("p", "3000", "port to serve on")

flag.Parse()

progSettings := LayoutStruct{"static", \*port, "SqlLedger"}

fs := http.FileServer(http.Dir("./static"))

http.Handle("/static/", http.StripPrefix("/static/", fs))

http.Handle("/", progSettings)

log.Printf("Serving %s on HTTP port: %s\n",

progSettings.StaticDir, progSettings.ListenPort)

log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", progSettings.ListenPort), nil))

}

1. Parameters can also be passed to the **HandleFunc()** function, but in this case, the second parameter must specify a function that returns the **func(ResponseWriter, \*Request)** function:

func CreateHTTPHandler(cnt LayoutStruct)

func(w http.ResponseWriter, r \*http.Request) {

return func(w http.ResponseWriter, r \*http.Request) {

lp := filepath.Join("templates", "layout.html")

fp := filepath.Join("templates", filepath.Clean(r.URL.Path))

info, err := os.Stat(fp)

if err != nil {

if os.IsNotExist(err) {

http.NotFound(w, r)

return

}

}

if info.IsDir() {

http.NotFound(w, r)

return

}

tmpl, err := template.ParseFiles(lp, fp)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

err = tmpl.ExecuteTemplate(w, "layout", cnt)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

}

}

Call this function in **http.HandleFunc**:

http.HandleFunc("/", CreateHTTPHandler(progSettings))

#### With GorillaMux

All requests must travel through this router's PathPrefix when you are working with a Gorilla router. The main function is slightly different:

func main() {

port := flag.String("p", "3000", "port to serve on")

flag.Parse()

progSettings := LayoutStruct{"static", \*port, "SqlLedger"}

route := mux.NewRouter()

fs := http.FileServer(http.Dir("./static/"))

**route.PathPrefix("/static/")**.Handler(http.StripPrefix("/static/", fs))

**route.PathPrefix("/app")**.HandlerFunc(CreateHTTPHandler(progSettings))

log.Printf("Serving %s on HTTP port: %s\n", progSettings.StaticDir,

progSettings.ListenPort)

log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", progSettings.ListenPort), route))

}

The **filepath.Clean(r.URL.Path)** will return **app/example.html** after sending the **localhost:3000/app/example.html** request. Therefore, you need to push the **example.html** file into the **app** subdirectory or hardcode a name of the file. The second choice simplifies logic of the handler:

func CreateHTTPHandler(cnt LayoutStruct) http.HandlerFunc {

return func(w http.ResponseWriter, r \*http.Request) {

fp := filepath.Join("templates", "example.html")

lp := filepath.Join("templates", "layout.html")

tmpl, err := template.ParseFiles(lp, fp)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

err = tmpl.ExecuteTemplate(w, "layout", cnt)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

}

}

Gorilla mux also allows you to use an object with the ServeHTTP function:

type LayoutStruct struct {

StaticDir string

ListenPort string

Subject string

}

func (cnt LayoutStruct) ServeHTTP(w http.ResponseWriter, r \*http.Request) {

lp := filepath.Join("templates", "layout.html")

fp := filepath.Join("templates", "example.html")

tmpl, err := template.ParseFiles(lp, fp)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

err = tmpl.ExecuteTemplate(w, "layout", cnt)

if err != nil {

log.Print(err.Error())

http.Error(w, http.StatusText(500), 500)

return

}

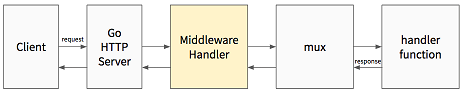
}

In this case, you specify name of this object in the routing table:

route.PathPrefix("/app").Handler(progSettings)

### Middleware handler

A middleware handler is an http.Handler that wraps another http.Handler to do some pre- and/or post-processing of the request. It's called "middleware" because it sits in the middle between the Go web server and the actual handler.



#### Name/password login

[How to correctly use Basic Authentication in Go – Alex Edwards](https://www.alexedwards.net/blog/basic-authentication-in-go" \l "protecting-a-web-application)

[Middleware Patterns in Go (drstearns.github.io)](https://drstearns.github.io/tutorials/gomiddleware/)

[Making and Using HTTP Middleware – Alex Edwards](https://www.alexedwards.net/blog/making-and-using-middleware)

## WEB Components

A Web Component is a way to create an encapsulated, single-responsibility code block that can be reused on any page. It plays the same role as a function in other programming languages. You can make a nice page header, name it **page-header** and use it on all pages just by writing a single element

<page-header></page-header>

The WEB component consists of three parts:

* Custom element,
* HTML template,
* Shadow DOM

#### Custom element

Here is an example of the hello-world custom element (file **HelloWorld.js**):

class HelloWorld extends HTMLElement {

connectedCallback() {

this.textContent = 'Hello World';

}

}

customElements.define('hello-world', HelloWorld);

Each custom element is an ES6 class that must be binded to the **CustomElementRegistry**:

customElements.define('hello-world', HelloWorld);

The first parameter of this operator specifies a name of the element. By convention, it must be two words joined by a minus sign. The second parameter is a name of the class. Again, by convention, one must use the same words as were used in a name of custom element.

Here is an example of the **Autonomous custom elements**, the class inherits from **HTMLElement** in this case. Loading an element into a page is exactly the same as any other JS file:

<!DOCTYPE html>

<html>

<head>

<title>Page Title</title>

**<script type="module" src="./helloworld.js"></script>**

</head>

<body>

<h1>This is an example of the custom element</h1>

**<hello-world></hello-world>**

</body>

</html>

**Type="module"** may be omitted in the **script** element, but then add the **deferred** attribute:

<script src="./helloworld.js" defer></script>

A single element can also be used:

<hello-world/>

but form of two elements allows you to pass additional information into the component:

<hello-world>This text will be passed to the component</hello-world>

WEB component allows you to create **Customized built-in elements**. The base class can be any other element in that case.

class WordCount extends HTMLParagraphElement {

constructor() {

// Always call super first in constructor

super();

// Element functionality written in here

}

}

customElements.define('word-count', WordCount, { extends: 'p' });

In this case it is necessary to specify the base element when registering the component (see the object in the third parameter). In an HTML page, you specify the **base element** with the **is** attribute:

<p is="word-count"></p>

The custom element may be created using JavaScript. Use

document.createElement("hello-world")

**Autonomous custom element** requires a bit different operator:

document.createElement("p", { is: "word-count" })

Note that the first parameter specifies the **base** element and object in the second parameter defines the actual element.

#### Lifecycle of the custom element

Examples in the previous paragraph show two very important methods of custom element: **constructor()** and **connectedCallback()**. These two methods are called at different time:

* **constructor()** is called when the element is created; you can create the Shadow DOM here, but you can't add Nodes inside the normal DOM, and you can't add or set an attribute either.
* **connectedCallback()** is called when (after) the element is attached to the DOM.

Definion

A custom element is defined when customElements.define is called:

class HelloWorld extends HTMLElement {

connectedCallback() {

this.textContent = 'Hello World';

}

}

customElements.define('hello-world', HelloWorld);

Defining an element doesn't trigger either the constructor or the connectedCallback methods since it does not create an instance of an element. **An element can only be defined once.**

Create

An element can be created in JavaScript in two ways:

// can happen before definition

const myElement = document.createElement('my-element');

// can only happen if already defined

const myElement = new MyElement();

Creation triggers the **constructor**, if the element has already been defined. The constructor is called once per element instance.

Insert

An element is inserted into the DOM imperatively with JS:

document.body.append(myElement);

Any HTML element may be used instead of **document.body.** Insertion triggers the connectedCallback method, if the element has already been defined. Here is a trap:

<html>

<head>

<title>Page Title</title>

<!-- <script type="module" src="./helloworld.js"></script> -->

<script src="./helloworld.js"></script>

</head>

<body>

<h1>This is an example of the custom element</h1>

<p id="content">

</p>

<!-- <hello-world></hello-world> -->

<script>

const myElement = new HelloWorld();

parent = document.getElementById("content");

parent.append(myElement);

</script>

</body>

</html>

This HTML page will show nothing if you use

<script type="module" src="./helloworld.js"></script>

The browser defers module parsing, thus the **hello-world** element is undefined during the second script execution.

**connectedCallback** is called when the element is inserted into the DOM.

**disconnectedCallback** method called each time it is removed from the DOM

Declare

An element is declared when parsed as HTML. It may be done with HTML or JavaScript:

<my-element></my-element>

document.body.innerHTML = '<my-element></my-element>';

Declaration triggers the constructor and connectedCallback methods, if the element has already been defined.

Upgrade

If the browser encounters any <**time-formatted**> elements before **customElements.define**, that’s not an error. But the element is yet unknown, just like any non-standard tag. Such “undefined” elements can be styled with CSS selector **:not(:defined)**.

When customElement.define is called, they are “upgraded”: a new instance of **TimeFormatted** is created for each, and **connectedCallback** is called. They become :defined.

#### CustomElementRegistry

The [CustomElementRegistry](https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry) interface provides methods for registering custom elements and querying registered elements. To get an instance of it, use the **window.customElements** property. Here is a list of instance methods

define

Defines a new [custom element](https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_custom_elements).

define(name, constructor)

define(name, constructor, options)

get

Returns the **constructor** for the named custom element, or undefined if the custom element is not defined.

let ctor = customElements.get("my-paragraph");

upgrade

The customElements **upgrade**() method upgrades all shadow-containing custom elements of the document in a node subtree, even before they are connected to the main document.

customElements.upgrade(root);

This method returns void. Parameter **root** is a node instance with shadow-containing descendant elements that are to be upgraded.

const el = document.createElement("spider-man");

class SpiderMan extends HTMLElement {}

customElements.define("spider-man", SpiderMan);

console.assert(!(el instanceof SpiderMan)); // not yet upgraded

customElements.upgrade(el);

console.assert(el instanceof SpiderMan); // upgraded!

whenDefined

The **whenDefined()** method of the **CustomElementRegistry** interface returns a Promise that resolves when the named element is defined. This interesting example of this function is provided on the [MDN](https://udn.realityripple.com/docs/Web/API/CustomElementRegistry/whenDefined) page. An asynchronous function hides the **Loading...** text when all menu items become defined.

#### Parameters

The data can be sent to the custom element using the element attributes:

The data can be sent to the element using the element attributes. It is possible to send scalar data or JSON

<!DOCTYPE html>

<html>

<head>

<title>Creating Custom Elements</title>

<script src="./indexOne.js" type="module"></script>

</head>

<body>

<salutation-element

name="Vladimir"

id=25

details='{"age": "20", "designation": "Developer"}'>

</salutation-element>

</body>

</html>

The custom control can read the parameters in the **connectedCallback** function:

class SalutationElement extends HTMLElement {

connectedCallback() {

const userName = this.attributes.name.value;

const userDetails = JSON.parse(this.attributes.details.value);

this.innerHTML = `<p>

Hello ${userName}<br/>

your age is ${userDetails}</p>`

}

}

customElements.define('salutation-element', SalutationElement);

A better example of using parameters is provided in the [github](https://github.com/mappmechanic/super-button-web-component/blob/master/Steps/tutorial1-Basic Custom Element.md) site. An example is build using a bit outdated constructions, so I'm providing a corrected version here.

File **super-button.js**:

class SuperButton extends HTMLElement {

get labelText() {

return this.getAttribute('label-text');

}

set labelText(value) {

if(value) {

this.setAttribute('label-text', value);

}

}

get color() {

return this.getAttribute('color');

}

set color(value) {

this.setAttribute('color', value);

}

static get observedAttributes() {

return ['label-text', 'color'];

}

connectedCallback() {

this.innerHTML = `

<div class="button">

${this.labelText}

</div>

<style>

.button {

width: 150px;

cursor: pointer;

border-radius: 10px;

text-align: center;

padding: 5px;

border: 1px solid #999;

box-shadow: 0px 4px 2px -2px;

}

.grey { background: #e5e5e5 }

.red { background: #ea5e4c }

.yellow { background: #ffc60e }

.blue { background: #93cefc }

</style>

`;

this.updateColor(this.color);

}

attributeChangedCallback(name, oldValue, newValue) {

switch(name) {

case 'label-text':

if(this.querySelector('.button') && newValue) {

this.querySelector('.button').textContent = newValue;

}

break;

case 'color':

this.updateColor(newValue, oldValue);

break;

}

}

updateColor(newValue, oldValue) {

const defaultColor = 'grey';

const allowedColors = ['red','yellow','blue'];

const newColor = allowedColors.indexOf(newValue) > -1 ? newValue : defaultColor;

const btnContainer = this.querySelector('.button');

if(btnContainer && oldValue) {

btnContainer.classList.remove(oldValue);

}

if(btnContainer && newColor) {

btnContainer.classList.add(newColor);

}

}

}

window.customElements.define('super-button', SuperButton);

Lines 2-18: all updatable attributes are defined as **getters** and **setters** of the ES6 class. The component can retrieve the getter’s value using the **this** pointer, see to the 27th line. The setter value can be changed both on the page where the component is declared and inside the component.

The static function **observedAttributes()** returns an array with the names of the tracked attributes. to which the component must respond, see lines 20-22. The component calls **attributeChangedCallback** function after every change of tracked value .

The **connectedCallback()** function (lines 24-46) describes a content and styles of the component. Possible, but not the best option. A better description will be provided after examining the **templates** and **shadowDom**.

The **attributeChangedCallback(name, oldValue, newValue)** function (lines 48-59) is called automatically after any tracked attribute has been changed. This function may call any other method of the component, see lines 56, 61-72.

File **index.html:**

<!DOCTYPE>

<html>

<head>

<title>Testing - Web Components</title>

<script type="module" src="./super-button.js"></script>

</head>

<body>

Button1: <super-button label-text="Super Button1"></super-button> <br />

Button2: <super-button label-text="Super Button2"></super-button>

<br /><br />

Update Button1 Label <br />

<input type="text" id="newLabel"><br />

<button onclick="updateBtn1Label()">Update Label</button>

<br /><br />

New Color <br />

<select id="newColor">

<option value="">Default</option>

<option value="red">Red</option>

<option value="yellow">Yellow</option>

<option value="blue">Blue</option>

<option value="green">Green</option>

</select> <br />

<button onclick="updateBtn2Color()">Update Color</button>

<script>

function updateBtn1Label() {

const newLabel = document.getElementById('newLabel').value;

const superBtns = document.getElementsByTagName('super-button');

superBtns[0].labelText = newLabel;

};

function updateBtn2Color() {

const newColor = document.getElementById('newColor').value;

const superBtns = document.getElementsByTagName('super-button');

superBtns[1].color = newColor;

};

</script>

</body>

</html>

WEB component is loaded with the link tag in the original version:

<link rel="import" href="./super-button.html">

HTML imports feature has been deprecated thus I changed it with script tag, see line 5. You can also use plain JavaScript with an attribute **deferred**. Loading a module gives you a lot of extra options.

Lines 8, 9 demonstrate how to declare a component by giving an initial value for the parameter .

Lines 11-14 with a helper function **updateBtn1Label()** updates attribute **label-text**. Direct call of the setAttribute was used in the original:

superBtns[0].setAttribute('label-text', newLabel);

Both variants are equivalent.

Lines 11-14 with helper function **updateBtn2Color()** upates attribute color. Once again, setter was used instead of direct call of **setAttribute** function.

#### Template

The **<template>** HTML element is a mechanism for holding HTML that is not to be rendered immediately when a page is loaded but may be instantiated subsequently during runtime using JavaScript. While the parser does process the contents of the **<template>** element while loading the page, it does so only to ensure that those contents are valid; the element's contents are not rendered, however. Corresponding **HTMLTemplateElement** interface has a standard **content** property (without an equivalent content/markup attribute), which is a read-only DocumentFragment containing the DOM subtree which the template represents. Let's examine this HTML snippet:

<table id="producttable">

<thead>

<tr>

<td>UPC\_Code</td>

<td>Product\_Name</td>

</tr>

</thead>

<tbody>

</tbody>

</table>

<template id="productrow">

<tr>

<td class="record"></td>

<td></td>

</tr>

</template>

This page will display an empty table, but it will have two rows after executing this JavaScript:

const tbody = document.querySelector("tbody");

**const template = document.querySelector("#productrow");**

**const clone = template.content.cloneNode(true);**

let td = clone.querySelectorAll("td");

td[0].textContent = "1235646565";

td[1].textContent = "Stuff";

tbody.appendChild(clone);

const clone2 = template.content.cloneNode(true);

td = clone2.querySelectorAll("td");

td[0].textContent = "0384928528";

td[1].textContent = "Acme Kidney Beans 2";

tbody.appendChild(clone2);

Events cannot be attached to a **DocumentFragment**, so make a copy of the **template.content** and attach events to that copy. See [MDN](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template) for an example of this action.

Templates can also be created with JavaScript:

const template = document.createElement('template');

template.innerHTML = `

<h1>Hello, World!</h1>

<p>And all who inhabit it</p>

`;

document.body.appendChild(template.content);

The latter variant is much better suited for building custom components, since it can be placed in the same file as code of the component. What's more, the template written this way can be placed in a separate file and loaded into the component using the **import** statement.

#### Shadow dom

An important aspect of web components is encapsulation — being able to keep the markup structure, style, and behavior hidden and separate from other code on the page so that different parts do not clash. The **Shadow DOM** API is a key part of this, providing a way to attach a hidden separated DOM to an element.

Shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a **shadow root**, underneath which you can attach any element, in the same way as the normal DOM.



Image was copied from "[Using shadow dom](https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM)" link. The following terminology is used in the figure:

* **Shadow host**: The regular DOM node that the shadow DOM is attached to.
* **Shadow tree**: The DOM tree inside the shadow DOM.
* **Shadow boundary**: the place where the shadow DOM ends, and the regular DOM begins.
* **Shadow root**: The root node of the shadow tree.

You can attach a shadow root to any element using the Element.attachShadow() method:

const shadowOpen = elementRef.attachShadow({ mode: "open" });

const shadowClosed = elementRef.attachShadow({ mode: "closed" });

If you attach a **shadow root** to a custom element with mode: **closed** set, you won't be able to access the shadow DOM from the outside — **myCustomElem.shadowRoot** returns null. Use this expression attching a shadow root inside the constructor of custom element:

const shadow = this.attachShadow({ mode: "open" }); // or {mode: "closed"}

The shadow dom may be also connected inside connectedCallback() function:

connectedCallback() {

const shadow = this.attachShadow({ mode: 'closed' });

shadow.innerHTML = `

<style>

p {

text-align: center;

font-weight: normal;

padding: 1em;

margin: 0 0 2em 0;

background-color: #eee;

border: 1px solid #666;

}

</style>

<p>Hello ${ this.name }!</p>`;

}

You can attach stylesheet with link element:

const linkElem = document.createElement("link");

linkElem.setAttribute("rel", "stylesheet");

linkElem.setAttribute("href", "style.css");

shadow.appendChild(linkElem);

An example of such custom element is presented on [GitHub](https://github.com/mdn/web-components-examples/blob/main/popup-info-box-external-stylesheet/main.js) site.

#### Slot

Slots are used inside templates of the custom elements:

<!-- Custom element -->

<template id="template-id">

<h2>Template with Slot</h2>

<p>

<slot **name**="slot-id">Named Slot</slot>

</p>

<p>

<slot **name**="sl-id">Named Slot without declaration</slot>

</p>

<p>

<slot>Unnamed slot</slot>

</p>

</template>

<!-- Host -->

<our-custom-element>

<span **slot="slot-id"**>This text will replace named slot</span>

<span>This text will replace unnamed slot</span>

</our-custom-element>

The text inside **slot** element is its default value the slot. The browser will display this text if the declaration of the custom element has no corresponding code.

## Custom events

Custom events can be created in two ways:

1. Using the **Event** constructor

const myEvent = new **Event**('myevent', {

bubbles: true,

cancelable: true,

composed: false

})

1. Using the **CustomEvent** constructor

const myEvent = new **CustomEvent**("myevent", {

detail: {},

bubbles: true,

cancelable: true,

composed: false,

});

Parameters:

* **bubbles** - the property specifies whether the event should be propagated upward to the parent element. The default value of this property is **false** - the custom event does not propagate. Set this property to **true**, then the custom event will behave exactly like native DOM events. You can stop the propagation of the event via event.stopPropagation().
* **cancelable** - the property specifies whether the event should be cancelable. Native DOM events are cancelable by default, so you can call **event.preventDefault()** on them, which will prevent the default action of the event. If the custom event has **cancelable** set to **false**, **calling event.preventDefault()** will not perform any action.
* **composed** - the property specifies whether an event should bubble across from the **shadow** **DOM** (created when using web components) to the real **DOM**. If **bubbles** is set to **false**, the value of this property won’t matter because you’re explicitly telling the event not to bubble upward. However, if you want to dispatch a custom event in a web component and listen on it on a parent element in the real DOM, then the **composed** property needs to be set to true.
* **detail** – any data that needs to be passed to the listener. Set a value to this property before performing the **dispatchEvent** operation.

Events can be dispatched to any object that extends **EventTarget**, and they include all **HTML** **elements**, the document, the window, etc.

const myEvent = new CustomEvent("myevent", {

detail: {},

bubbles: true,

cancelable: true,

composed: false,

});

document.querySelector("#someElement").dispatchEvent(myEvent);

To listen for the custom event, add an event listener to the element you want to listen on, just as you would with native DOM events.

document.querySelector("#someElement").addEventListener("myevent", (event) => {

console.log("I'm listening on a custom event");

});

You have to set correct value of the this pointer when calling a method inside of the object. Use a **lambda expression** and the **bind()** function for doing that:

var myButton = {

content: 'OK',

click() {

console.log(this.content + ' clicked');

}

hookEvent(element) {

// Use bind() to ensure 'this' is the 'this' inside click()

element.addEventListener('click', this.click.bind(this));

}

};

Lambda expressions do not change the this pointer, so you may not need to use bind():

element.addEventListener('click', () => this.click());

You can also add extra parameters after the 1st (this) parameter and bind will pass in those values to the original function. Any additional parameters you later pass to the bound function will be passed in after the bound parameters:

var sum = function(a, b) {

return a + b;

};

var add5 = sum.bind(null, 5);

console.log(add5(10));

Value of **this** pointer may be changed by **call** and **apply** functions. Both methods are parts of the function prototype:

call(thisArg)

call(thisArg, arg1, /\* …, \*/ argN)

apply(thisArg)

apply(thisArg, argsArray)

[Custom events in JavaScript: A complete guide - LogRocket Blog](https://blog.logrocket.com/custom-events-in-javascript-a-complete-guide/)

[What is Lit? – Lit](https://lit.dev/docs/)

## LIT

Lit is a simple library for building fast, lightweight web components. Every Lit component is a standard web component. Lit components can be used across multiple apps and sites, even if those apps and sites are built on a variety of front-end stacks. Each Lit component is a self-contained unit of UI, assembled from smaller building blocks: standard HTML elements and other web components. In turn, each Lit component is itself a building block that can be used–within an HTML document, another web component, or a framework component–to build larger and more complex interfaces. LIT components may be written in TypeScript or JavaScript.

[Introduction · Bootstrap v5.0 (getbootstrap.com)](https://getbootstrap.com/docs/5.0/getting-started/introduction/)

[Bootstrap 5 Tutorial (w3schools.com)](https://www.w3schools.com/bootstrap5/index.php)

## Bootstrap 5

Bootstrap is the world’s most popular framework for building responsive, mobile-first sites. Use jsDelivr for installation CSS and JS into the page.

<!DOCTYPE html>

<html lang="en">

<head>

<title>Bootstrap 5 Example</title>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

...

<script

src="[https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js](https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/js/bootstrap.bundle.min.js)">

</script>

</body>

</html>

File **bootstrap.bundle.min.js** includes **Popper** for tooltips and popovers.

The main differences between Bootstrap 5 and Bootstrap 3 & 4, is that Bootstrap 5 has switched to vanilla JavaScript instead of jQuery. Bootstrap 5 also requires a containing element to wrap site contents. Bootstrap 5 is designed to be responsive to mobile devices. Mobile-first styles are part of the core framework.

## W3.CSS

W3.CSS is a modern, responsive, mobile first CSS framework. It works with:

* all browsers: Chrome. Firefox. Edge. IE. Safari. Opera.
* -all devices: Desktop. Laptop. Tablet. Mobile.

W3.CSS is standard CSS only (No jQuery or JavaScript library). To use W3.CSS in your web site, just add a link to "w3.css" from your web pages:

<!DOCTYPE html>

<html lang="en">

<head>

<title>Bootstrap 5 Example</title>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

**<link href="https://www.w3schools.com/w3css/4/w3.css" rel="stylesheet">**

</head>

<body>

...

</body>

</html>

or download w3.css from [w3css\_downloads](https://www.w3schools.com/w3css/w3css_downloads.asp) and add a link to w3.css in the head of your HTML. You have to use the request multiplexer and process CSS file as static resource. The process is explained in the "**Serving Static Files and Sites**" topic of this document. The link will be

<link rel="stylesheet" href="public/css/w3 .css">

if a path to w3.css is /public/css/ w3.css.

[Docker with CGI perl. hello world tutorial | by lojorider lojo | Medium](https://medium.com/@lojorider/docker-with-cgi-perl-a4558ab6a329)

[SQL-Ledger ERP](https://www.sql-ledger.com/cgi-bin/nav.pl?page=source/index.html&title=Download) (download)

[SQL-Ledger ERP](https://www.sql-ledger.com/cgi-bin/nav.pl?page=source/readme.txt&title=README) (install)

[step\_by\_step\_installation:ubuntu [International SQL-Ledger Network Association] (sql-ledger-network.com)](https://www.sql-ledger-network.com/dokuwiki/doku.php?id=step_by_step_installation:ubuntu)

[Tutorial - Perl CGI on Apache [ Step by step ] (techexpert.tips)](https://techexpert.tips/apache/perl-cgi-apache/) //veikia

[Ubuntu 18.04 LTS : Apache2 : Use Perl Scripts : Server World (server-world.info)](https://www.server-world.info/en/note?os=Ubuntu_18.04&p=httpd&f=2)

[GitHub - Tekki/sql-ledger: :books: The official SQL-Ledger version, plus some additions.](https://github.com/Tekki/sql-ledger)

[bootstrap CDN by jsDelivr - A CDN for npm and GitHub](https://www.jsdelivr.com/package/npm/bootstrap)

https://www.alexedwards.net/blog/serving-static-sites-with-go

## WebPack

**WebPack** examines all of the modules in your application, creates a dependency graph, then intelligently puts all of them together into one or more bundle(s) that your index.html file can reference. Importantly, **WebPack** not only collects all the files into one big file, but also arranges them in the right order. I.e. if File a.xxx has a reference to file b.xxx, then b.xxx will be placed before a.xxx in the generated file. WebPack can also perform the necessary transformations: SASS/LESS to regular CSS, TypeScript to JS, and so on.

#### Installation

sudo apt -y install webpack

**WebPack** is launched via **node**, so a sufficient number of packages will be installed additionally. Windows users using **WebPack** using **npm**:

npm install webpack webpack-cli –save-dev

Check presence of the **npm** package and install it if you don't have **npm:**

npm –version

sudo apt install npm

https://ui.dev/webpack

## Material design