# 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)**)

}

The main work in this program is done by the function [ListenAndServe](https://pkg.go.dev/net/http" \l "ListenAndServe). The address ":8080" is a shortened form of the "localhost:8080".

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:

https://sathishvj.medium.com/web-handlers-and-middleware-in-golang-2706c2ecfb75

https://medium.com/@chrisgregory\_83433/chaining-middleware-in-go-918cfbc5644d

https://www.digitalocean.com/community/tutorials/how-to-make-an-http-server-in-go

## 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/)