



WEBCOMPONENTS.ORG

Building blocks for the web

Use
The
Platform

Modulares Web

ES6 Module, lit-html, LitElement,
Web Components, Komponenten-Marktplätze

Modulares Web

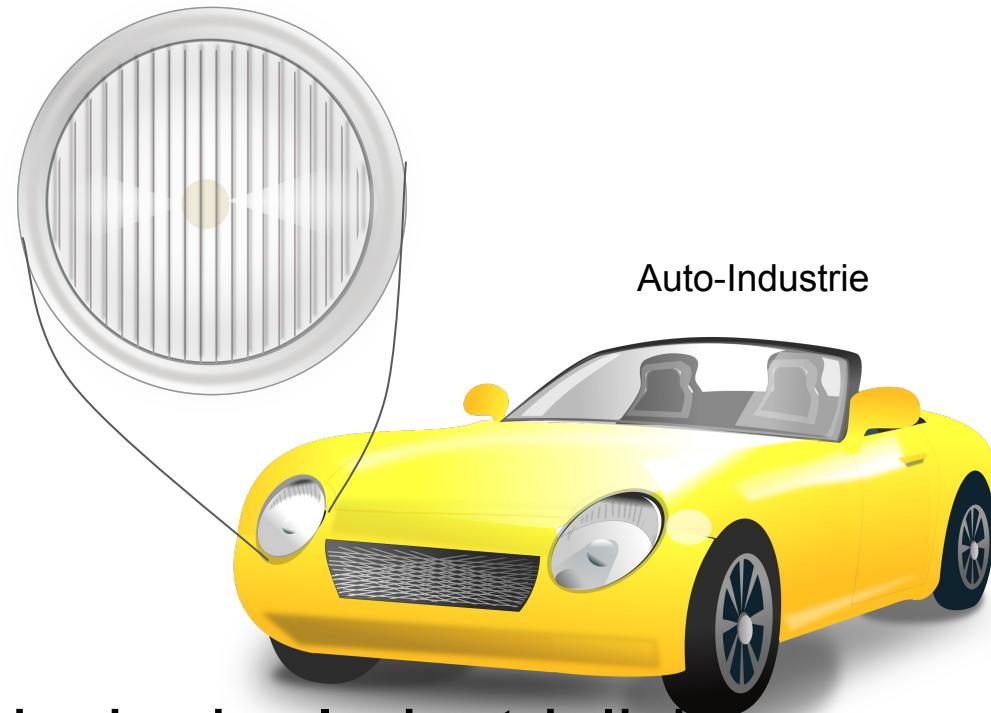
- Modulares JavaScript ⇒ ES6 module import & export
- Modulares HTML ⇒ Custom Elements
- Modulares CSS ⇒ Shadow DOM



Use
The
Platform

Ausgangsfrage: Warum überhaupt Modularität?

- Professionalisierung
- Arbeitsteilung
 - Taylorismus
 - Spezialisten
- Modul-Industrie
 - Autos aus Modulen



⇒ Modularität als Prinzip der Industrialisierung

Woran erkennt man den Reifegrad einer Industrie?

- Das Rad wird nicht mehr neu erfunden.
- Modularisierung
- Arbeitsteilung, Spezialisierung
- Markt
 - Angebot und Nachfrage
 - ⇒ Komplexität

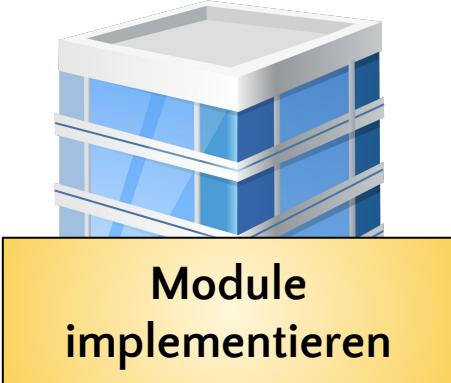


Wie reif ist die Informatik-Industrie?

Arbeitsteilung

"select the right person for the right job"

Entwickler



*funktional
objekt-orientiert*

Designer



*deklarativ
HTML-Spracherweiterung*



Frederic W. Taylor (1856-1915)
Quelle: [Wikipedia](#)
"select the right person
for the right job"

Gliederung

1. JavaScript-Module (ES6 module)
2. Web-Komponenten
3. lit-html
4. LitElement
5. Marktplätze für Komponenten

ES6 Modules

```
<script type="module">
```

import

export

module-import.html

```
<!DOCTYPE html>
<script type="module">
  import { hello, logger } from "./exported_module.mjs";
  logger( hello , " import" );
</script>
```

Konvention ".mjs" statt ".js"

exported_module.mjs

```
export const hello = "Hello";

export function logger( ...param ){
  console.log( ...param );
}
```

<https://jakearchibald.com/2017/es-modules-in-browsers/>

<https://v8.dev/features/modules>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>

ES 6 Import Syntax

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { export1 , export2 } from "module-name";
import { foo , bar } from "module-name/path/to/specific/un-exported/file";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export1 [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
var promise = import("module-name");
```

ES6 import: default

You could also export a *default* value from a module:

```
// lib.mjs
export default function(string) {
  return `${string.toUpperCase()}!`;
}
```

Such `default` exports can be imported using any name:

```
// main.mjs
import shout from './lib.mjs';
//           ^^^^^^
```

ES6 import

- ES6 modules are stored in files
- one module per file
- one file per module
- rekursives Laden
 - Module dürfen auch selbst wieder andere Module importieren
- nur einmal laden
- zirkuläre imports führen *nicht* zu Endlosschleifen

```
<!doctype html>
<script type="module">
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { export1, export2 } from "module-name";
import { foo , bar } from "module-name/path/to/specific/file";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export1 [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
const promise = import("module-name");
</script>
```

<https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/import>

Some restrictions apply to module specifiers in browsers. So-called “bare” module specifiers are currently not supported. This restriction is [specified](#) so that in the future, browsers can allow custom module loaders to give special meaning to bare module specifiers like the following:

```
// Not supported (yet):
import {shout} from 'jquery';
import {shout} from 'lib.mjs';
import {shout} from 'modules/lib.mjs';
```



On the other hand, the following examples are all supported:

```
// Supported:
import {shout} from './lib.mjs';
import {shout} from '../lib.mjs';
import {shout} from '/modules/lib.mjs';
import {shout} from 'https://simple.example/modules/lib.mjs';
```



For now, module specifiers must be full URLs, or relative URLs starting with `/`, `./`, or `../`.

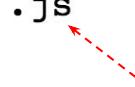
<https://v8.dev/features/modules>

Unterschiede <script> und <script type="module">

- Modules have strict mode enabled by default.
 - "use strict"; ⇒ https://www.w3schools.com/js/js_strict.asp
 - https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Strict_mode
- Modules have a **lexical top-level scope**. This means that for example, running `var foo = 42;` within a module does *not* create a global variable named `foo`, accessible through `window.foo` in a browser, although that would be the case in a classic script.
- The new static `import` and `export` syntax is only available within modules
 - it doesn't work in classic scripts.
- Because of these differences, **the same JavaScript code might behave differently when treated as a module vs. a classic script**. As such, the JavaScript runtime needs to know which scripts are modules.



Vergleich <script> und ES6 module

	Scripts	Modules
HTML element	<script>	<script type="module">
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of <code>this</code> at top level	window	undefined
Executed	synchronously	asynchronously
Declarative imports (<code>import</code> statement)	no	yes
Programmatic imports (Promise-based API)	yes	yes
File extension	.js	.js  .mjs vorgeschlagen

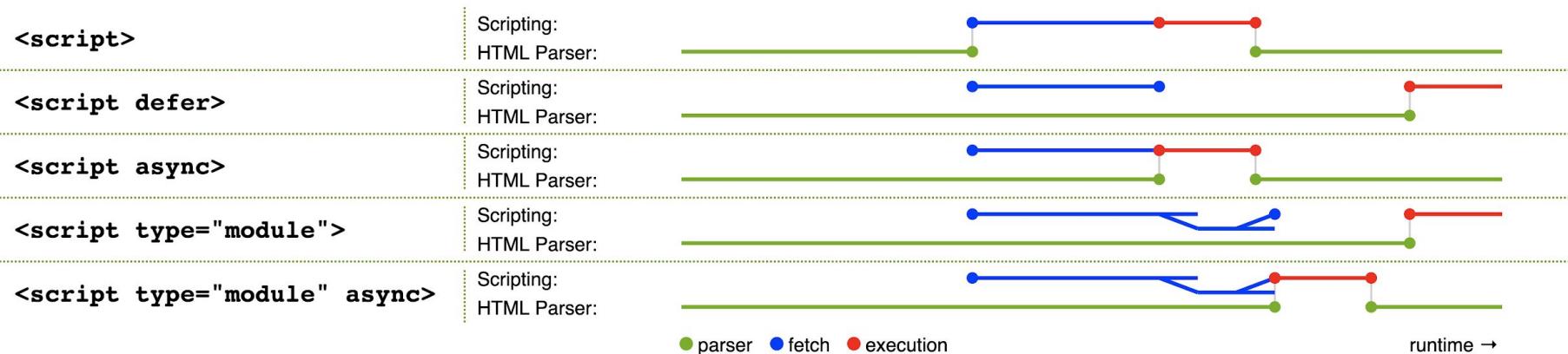
<script type="module" async defer>

Modules are deferred by default

to defer

aufschlieben; verschieben;
verzögern; zurückstellen {vt}

Classic <script>s **block** the HTML parser by default. You can work around it by adding **the defer attribute**, which ensures that the script download happens **in parallel** with HTML parsing.



Modules are deferred by default

<https://v8.dev/features/modules>

Dynamic import()

The `import` keyword may be called as a function to dynamically import a module. When used this way, it returns a promise.

```
1 | import('/modules/my-module.js')
2 |   .then((module) => {
3 |     // Do something with the module.
4 |  });
```

This form also supports the `await` keyword.

```
1 | let module = await import('/modules/my-module.js');
```

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import>

<https://developers.google.com/web/updates/2017/11/dynamic-import>

Gliederung

1. JavaScript-Module (ES6 module)
2. **Web-Komponenten**
3. lit-html
4. LitElement
5. Marktplätze für Komponenten

Vier W3C-Standards für Web-Komponenten

1. ES6 import
2. Custom Element
3. Shadow DOM
4. Template Element

Web Components Standards History v0 and v1

Wenn Sie Beispiele im Web finden, achten Sie zuerst darauf, ob v0 oder v1:

Unterschiede siehe:

<https://hayato.io/2016/shadowdomv1/>

	v0	v1
seit Datum	seit 2014-07-15	seit 2016-10-19
Chrome Version	ab Chrome 36+	ab Chrome 54+
Custom Elements API	<code>document.registerElement()</code>	<code>customElements.define()</code>
JavaScript	ES5	ES6

<https://developers.google.com/web/fundamentals/getting-started/primers/customelements#historysupport>

https://en.wikipedia.org/wiki/Google_Chrome_version_history

Custom Elements v1

- HTML-Spracherweiterung:
eigene Tags definieren
- <my-tag></my-tag>
 - immer mit Bindestrich
 - immer mit schließendem Tag
 - keine *self-closing tags*
 - jeden <Tag> nur einmal definieren
(sonst DOMException)
- Browser-Weiche
 - 'customElements' in window ⇒ v1
 - 'registerElement' in document ⇒ v0
 - sonst Polyfills
 - <https://github.com/webcomponents/custom-elements>

The diagram shows the creation of a new HTML element, `<toggle-color>`. It consists of two main parts: a code editor and a browser preview.

Code Editor:

```
<!DOCTYPE html>
<script>
class MyColor extends HTMLElement {
  constructor(){
    super(); super()
    this.color = false;
    this.addEventListener('click', (e) => {
      this.color = ! this.color; // true => false
      this.style.color = this.color ? "red" : "black";
    });
  }
}
window.customElements.define('toggle-color', MyColor);
</script>

<toggle-color><h1>Click me!</h1></toggle-color>
```

Annotations:

- `super()`: super() immer an 1. Stelle
- `this`: this === MyColor HTMLElement im DOM
- `<toggle-color>`: neues HTML-Tag

Browser Preview:

Two browser windows show the result of the code execution. Both windows display the text "Click me!" in large red font. A yellow callout points from the word "neues HTML-Tag" in the annotations to the newly created element in the browser's DOM tree.

Spec. § 4.13.2 Requirements for custom element constructors:



When authoring custom element constructors, authors are bound by the following conformance requirements:

- A parameter-less call to `super()` must be the **first statement in the constructor body**, to establish the correct prototype chain and `this` value before any further code is run.
- A return statement must not appear anywhere inside the constructor body, unless it is a simple early-return (return or return `this`).
- The constructor must not use the `document.write()` or `document.open()` methods.
- **The element's attributes and children must not be inspected, as in the non-upgrade case none will be present, and relying on upgrades makes the element less usable.**
- **The element must not gain any attributes or children, as this violates the expectations of consumers who use the `createElement` or `createElementNS` methods.**
- In general, work should be deferred to `connectedCallback` as much as possible—especially work involving fetching resources or rendering. However, note that `connectedCallback` can be called more than once, so any initialization work that is truly one-time will need a guard to prevent it from running twice.
- In general, the constructor should be used to **set up initial state and default values**, and to set up event listeners and possibly a shadow root.

Several of these requirements are checked during element creation, either directly or indirectly, and failing to follow them will result in a custom element that cannot be instantiated by the parser or DOM APIs. This is true even if the work is done inside a constructor-initiated microtask, as a microtask checkpoint can occur immediately after construction.

Custom Elements dürfen am Anfang keine Kinder haben

- Fehler durch `this.innerHTML = "Hello World";`
- "The element must not gain any attributes or children, as this violates the expectations of consumers who use the `createElement` or `createElementNS` methods."

["Requirements for custom element constructors"](#)
section of the spec

```
<script>

class MyElement extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = "Hello World";
  }
}
window.customElements.define('my-tag', MyElement);

</script>

<my-tag></my-tag>
```



DOMException: Failed to execute
'createElement' on 'Document': The result
must not have children

Custom Elements dürfen keine Attribute im Konstruktor benutzen

- Fehler durch `this.className = "class1"`
- "The element must not gain any attributes or children, as this violates the expectations of consumers who use the createElement or createElementNS methods."

["Requirements for custom element constructors"](#)
section of the spec

```
<!doctype html>
<h1>SquareLetter</h1>
<script>
class SquareLetter extends HTMLElement {
  constructor() {
    super();
    this.className = "class1"
  }
}
customElements.define("square-letter", SquareLetter);
</script>

<square-letter></square-letter>
```



Uncaught DOMException: Failed to construct 'CustomElement': The result **must not have attributes**

Lösung: Kinder in `connectedCallback()` hinzufügen

- In general, work should be deferred to `connectedCallback` as much as possible—especially work involving fetching resources or rendering.
However, note that `connectedCallback` can be called *more than once*, so any initialization work that is truly one-time will need a guard to prevent it from running twice.
- That is, add children in the connected callback.

["Requirements for custom element constructors"](#)
section of the spec



```
<script>
  class MyElement extends HTMLElement {
    connectedCallback() {
      this.innerHTML = "<h1>Hello World</h1>";
      this.className = "class1";
    }
  }
  window.customElements.define('my-tag', MyElement);
</script>

<my-tag></my-tag>
```



Aufgabe Counter

- soll anzeigen, wie oft geklickt wurde
- hat eine Zahl als Kind-Element
- soll universell einsetzbar sein

```
<!DOCTYPE html>

<script>
customElements.define( 'simple-counter',
    class extends HTMLElement {
        constructor(){
            super();
            this.counter = 0;
            this.addEventListener( 'click',
                () => this.textContent = ++this.counter );
        }
        connectedCallback(){
            this.textContent = this.counter;
        }
    });
</script>

<simple-counter></simple-counter>
```

neues HTML-Tag

Spec §4.13.6 Important Custom element reactions

Name	Called when
constructor()	An instance of the element is created or upgraded . Useful for initializing state, setting up event listeners, or creating a shadow dom . See the spec for restrictions on what you can do in the constructor.
connectedCallback()	Called every time the element is inserted into the DOM. Useful for running setup code, such as fetching resources or rendering. Generally, you should try to delay work until this time.
disconnectedCallback()	Called every time the element is removed from the DOM. Useful for running clean up code.
attributeChangedCallback(attrName, oldVal, newVal)	Called when an observed attribute has been added, removed, updated, or replaced. Also called for initial values when an element is created by the parser, or upgraded . Note: only attributes listed in the observedAttributes property will receive this callback.

observedAttributes

attributeChangedCallback(attr, oldVal, newVal)

- Attribute in HTML
- Properties in JS
- static *observedAttributes* ist die Liste aller Attribute, die beobachtet werden sollen



```
<script>
customElements.define( 'simple-counter',
  class extends HTMLElement {
    static get observedAttributes(){ return [ "start" ] };
    constructor(){
      super();
      this.counter = 0; // init property
      this.addEventListener( 'click',
        () => this.textContent = ++this.counter );
    }
    connectedCallback(){
      this.textContent = this.counter;
    }
    attributeChangedCallback( attr, oldVal, newVal ){
      if ( oldVal !== newVal ) {
        this.textContent = newVal; // Kinder-Inhalte ändern
        this.counter = parseInt( newVal ); // Properties ändern
      }
    }
  });
</script>
```

nur bei Prop. 'start' wird
attributeChangedCallback
aufgerufen

```
<simple-counter start="100"></simple-counter>
```

Update Kinderelemente, Attribute & Properties separat

```
<script>  
class MyColorCounter extends HTMLElement {  
  static get observedAttributes(){ return ['start']; }  
  constructor(){  
    super();  
    this.counter = 0;  
    this.color = false;  
    this.addEventListener('click', () => {  
      this.color = !this.color; // true => false  
      this.style.color = this.color ? "red" : "black";  
      this.counter += 1;  
      this.counterElement.textContent = this.counter;  
    });  
  } connectedCallback(){...} attributeChangedCallback(){...}  
}  
window.customElements.define('toggle-color', MyColorCounter);  
  
</script>  
  
<toggle-color start="10"><h1>Click me!</h1></toggle-color>
```

```
connectedCallback(){  
  this.counter = parseInt( this.getAttribute('start') );  
  this.innerHTML += `<h1>Anzahl Klicks:  
    <span id="counter">${this.counter}</span></h1>`;  
  this.counterElement = this.querySelector('#counter');  
}  
attributeChangedCallback( attr, oldVal, newVal ){  
  if (oldVal !== newVal) {  
    this.counter = parseInt( newVal );  
    if ( this.counterElement ) this.counterElement.textContent = this.counter;  
  }  
}
```

Property

Inhalt Kinderelement

<https://developers.google.com/web/fundamentals/web-components/custom-elements>



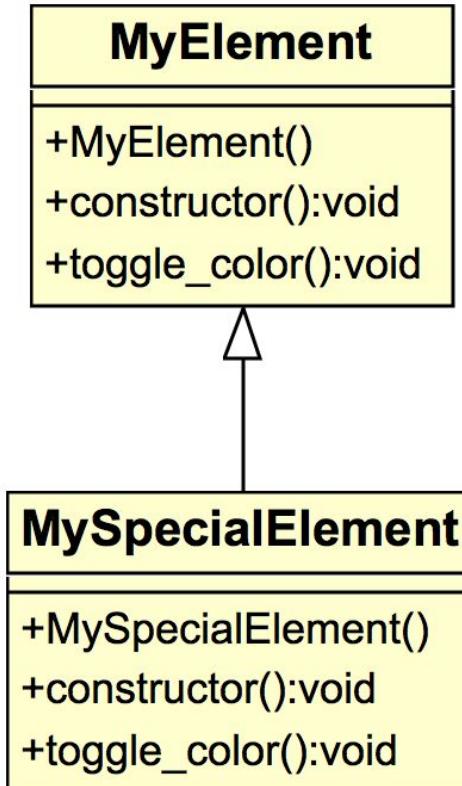
4.13.6 All Custom element reactions

- When upgraded, its constructor is run, with no arguments.
- When it becomes connected, its `connectedCallback` is called, with no arguments.
- When it becomes disconnected, its `disconnectedCallback` is called, with no arguments.
- When it is adopted into a new document, its `adoptedCallback` is called, given the old document and new document as arguments.
- When any of its attributes are changed, appended, removed, or replaced, its `attributeChangedCallback` is called, given the attribute's local name, old value, new value, and namespace as arguments. (An attribute's old or new value is considered to be null when the attribute is added or removed, respectively.)
- When the user agent resets the form owner of a form-associated custom element and doing so changes the form owner, its `formAssociatedCallback` is called, given the new form owner (or null if no owner) as an argument.
- When the form owner of a form-associated custom element is reset, its `formResetCallback` is called.
- When the disabled state of a form-associated custom element is changed, its `formDisabledCallback` is called, given the new state as an argument.
- When user agent updates a form-associated custom element's value on behalf of a user, its `formStateRestoreCallback` is called, given the new value and a string indicating a reason, "restore" or "autocomplete", as arguments.

Wie kann man Komponenten anpassen? Implementation Inheritance for Customizing

```
class MySpecialElement extends MyElement {  
    constructor() {  
        super();  
    }  
    toggle_color() {  
        this.color = ! this.color; // true => false  
        this.style.color = this.color ? "yellow" : "green";  
    }  
}  
  
customElements.define('my-tag', MySpecialElement);
```

Einfache Anpassbarkeit: Das Verhalten nur dort abändern, wo es nötig ist.



Shadow DOM introduces **scoped styles** to the web platform

```
<script>
customElements.define('my-shadowdom', class extends HTMLElement {
  constructor() {
    super();
    // Attach a shadow root to the element.
    let shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML =
      <style>
        p {
          color: red;
          border: thin solid;
          margin: 0.2rem;
        }
      </style>
      <p><b>Text inside Shadow DOM</b></p>
    ;
  }
});
</script>
```

Text outside of custom element

Text inside Shadow DOM

Text outside of custom element

The diagram illustrates the flow of styles from the main script to the shadow DOM. A dashed red arrow points from the style block in the main script to the style block within the shadow DOM. Another dashed red arrow points from the p element in the main script to the p element in the shadow DOM.

```
<style>
p {
  background-color: lightgreen;
  border: thin solid;
  margin: 0.2rem;
}
</style>
<p>Text outside of custom element</p>
<my-shadowdom>
  <p>Text inside of custom element (in Light DOM)</p>
</my-shadowdom>
<p>Text outside of custom element</p>
```

Open and Closed Shadow Root

- → Scopes for Custom Elements
- {mode: 'open'}
 - von außen Zugriff erlaubt
- {mode: 'closed'}
 - shadowRoot will return null

```
<!DOCTYPE html>
<html>
  <head></head>
... ▼<body> == $0
  <h1>Custom Element with Shadow DOM</h1>
  ▶<script>...</script>
  ▶<hello-world>
    ▶#shadow-root (open)
      <style>h1 { color: red; }</style>
      <h1>Hello World!</h1>
    </hello-world>
  </body>
</html>
```

// Use custom elements API v1 to register a new HTML tag and define its JS behavior
// using an ES6 class. Every instance of <hello-world> will have this same prototype.

```
<!doctype html>
<h1>Custom Element with Shadow DOM</h1>
<script>
customElements.define('hello-world', class extends HTMLElement {
  constructor() {
    super();
    // Attach a shadow root to <hello-world>.
    const shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML =
      <style>h1 { color: red; }</style>
      <h1>Hello World!</h1>
    `; // style is scoped to hello-world
  }
});
</script>

<hello-world></hello-world>
```

Anonymous ES6 class

Backtick for ES6 template string

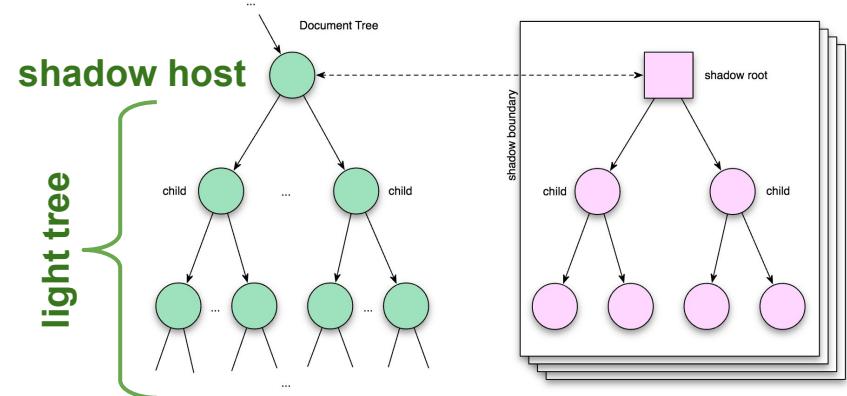
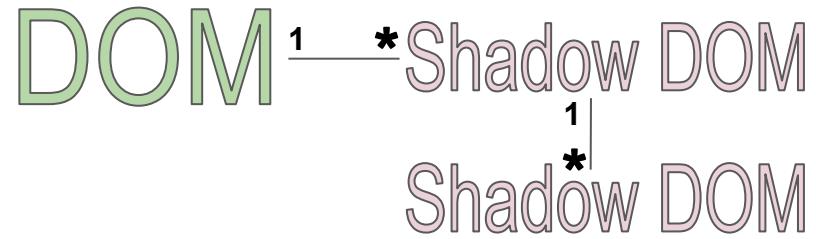
Custom Element with Shadow DOM

Hello World!

Shadow Tree - Definitions

- A shadow tree is a node tree whose root is a shadow root.
- A shadow root is always attached to another node tree through its shadow host. A shadow tree is therefore never alone. The node tree of a shadow root's host is sometimes referred to as the light tree.
- A shadow tree's corresponding light tree can be a shadow tree itself.
- An element is **connected** if its shadow-including root is a document.

<http://w3c.github.io/webcomponents/spec/shadow/>

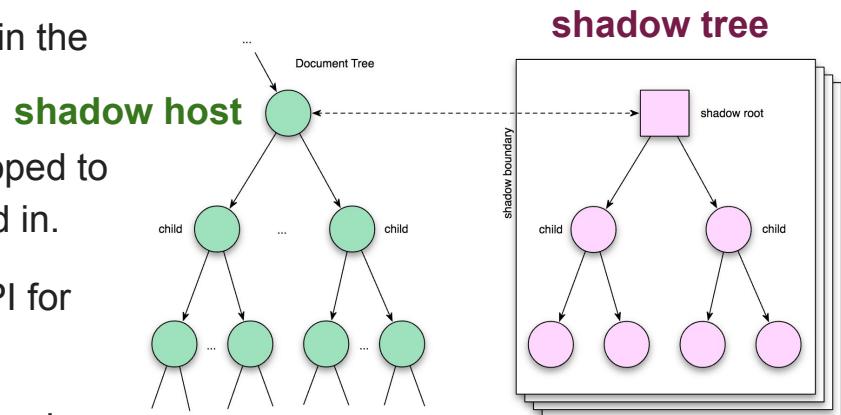


<https://www.w3.org/TR/2013/WD-components-intro-20130606/#shadow-dom-section>

Haupt-DOM

Vorteile des Shadow DOM

- **Isolated DOM:** A component's DOM is self-contained (e.g. `document.querySelector()` won't return nodes in the component's shadow DOM).
- **Scoped CSS:** CSS defined inside shadow DOM is scoped to it. Style rules don't leak out and page styles don't bleed in.
- **Composition:** Design a declarative, markup-based API for your component.
- **Simplifies CSS** - Scoped DOM means you can use simple CSS selectors, more generic id/class names, and not worry about naming conflicts.
- **Productivity** - Think of apps in chunks of DOM rather than one large (global) page.



<https://www.w3.org/TR/2013/WD-components-intro-20130606/#shadow-dom-section>

Slots als Platzhalter, wo der Light DOM in den Shadow DOM kommt

```
<script>
customElements.define('my-shadowdom', class extends HTMLElement {
  constructor() {
    super();
    // Attach a shadow root to the element.
    let shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML =
      <style>
        p {
          color: red;
          border: thin solid;
          padding: 0.2rem;
          margin: 0.2rem;
        }
      </style>
      <p><b>Text inside Shadow DOM</b></p>
      <p><slot></slot></p>
    `;
  }
});
</script>
```

Text outside of custom element

Text inside Shadow DOM

Text inside of custom element (in Light DOM)

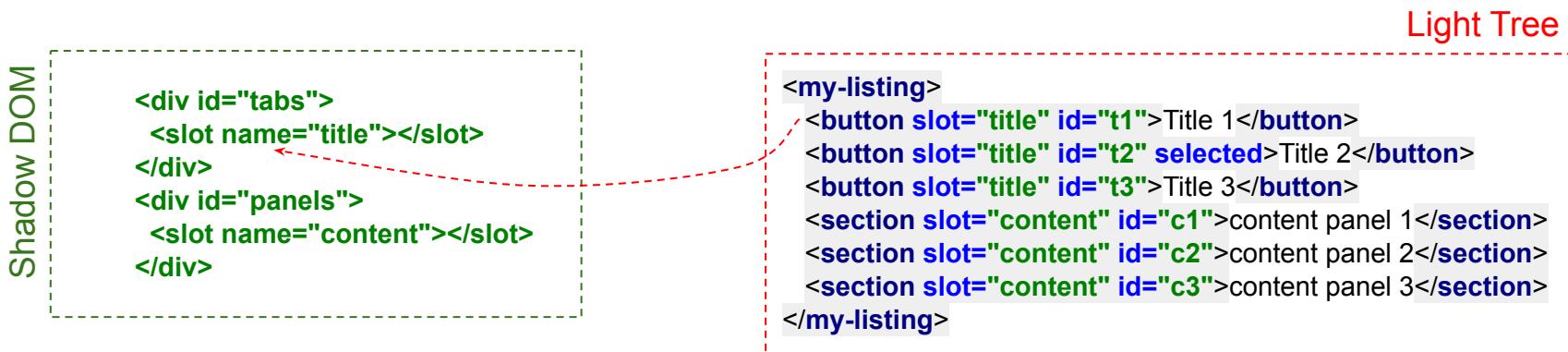
Text outside of custom element

```
<style>
  p {
    background-color: lightgreen;
    border: thin solid;
    margin: 0.2rem;
    padding: 0.2rem;
  }
</style>
```

```
<p>Text outside of custom element</p>
<my-shadowdom>
  <p>Text inside of custom element (in Light DOM)</p>
</my-shadowdom>
<p>Text outside of custom element</p>
```

Named Slots als Platzhalter im Shadow DOM

The **HTML `<slot>` element**—part of the [Web Components](#) technology suite—is a placeholder inside a web component that you can fill with your own markup, which lets you create separate DOM trees and present them together.



Named Slots for Shadow DOM Composition

Deklarativ in HTML statt imperativ in JavaScript

```
<script>
customElements.define('my-listing', class extends HTMLElement {
  constructor() {
    super(); // always call super() first in the ctor.
    const shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML =
      <style>#panels { border-style: solid;}</style>
      <div id="tabs">
        <slot name="title"></slot> ←
      </div>
      <div id="panels">
        <slot name="content"></slot> ←
      </div>
    ;
    this.addEventListener('click', e => {
      console.log( e.target.id );
    });
  }
});
</script>
```

Definition

Shadow DOM with slots

Title 1 Title 2 Title 3

content panel 1
content panel 2
content panel 3

Anwendung

```
<my-listing>
  <button slot="title" id="t1">Title 1</button>
  <button slot="title" id="t2" selected>Title 2</button>
  <button slot="title" id="t3">Title 3</button>
  <section slot="content" id="c1">content panel 1</section>
  <section slot="content" id="c2">content panel 2</section>
  <section slot="content" id="c3">content panel 3</section>
</my-listing>
```

Light Tree

Im Light Tree werden die slot-Werte spezifiziert, die in die slot-Platzhalter im Shadow Tree eingefügt werden.

<slot> ohne name für den Rest

<https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom>

Wie kann man das Web-Komponenten-Layout parametrisieren?

CSS Custom Properties

- Deklaration mit --name
- Benutzung mit var(--name)
- Kaskade mit var(x,y,z)
 - erst x, dann y, dann z

kein Teil des Web Components - Standards, aber gute Ergänzung
ohne Web-Komponenten

```
:root {  
  --main-color: #06c;  
}  
  
h1 {  
  color: var(--main-color);  
  background-color: var(--main-color, blue);  
}
```

https://developer.mozilla.org/en-US/docs/Web/CSS/--*

<https://www.w3.org/TR/css-variables/>

<https://developers.google.com/web/updates/2016/02/css-variables-why-should-you-care>

mit Web-Komponenten

```
/* Outside Web Component */  
fancy-tabs {  
  margin-bottom: 32px;  
  --background-color: black;  
}
```

Parameter setzen

```
// Inside Web Component  
// Create shadow DOM for the component.  
let shadowRoot = this.attachShadow({mode: 'open'});  
shadowRoot.innerHTML = `  
  <style>  
    :host {  
      background: var(--background-color, #9E9E9E);  
    }  
  </style>
```

Parameter definieren

Das HTML <template> - Element

- <template> ist ein zusätzliches HTML-Tag, das zum W3C Web Components Standard gehört.
- Mit dem Template-Element kann man HTML-Dokument-Fragmente deklarieren und vorbereiten, die beim Laden noch *nicht* gerendert werden.
- Sie werden geklont und in den DOM per Skript eingefügt. Erst dann werden sie "live" geschaltet.
- Vorteile:
 - DRY ⇒ Wiederverwendung
 - Performanz

```
<!doctype html>
<h1>W3C HTML Template</h1>
<div id="main"></div>
<template id="template">
  <div>Hello from <span>
  class="name">Template</span>
  </div>
</template>
```

```
<script>
var host = document.querySelector('#main');
var template = document.querySelector('#template');
host.appendChild(template.content.cloneNode(true));
host.appendChild(template.content.cloneNode(true));
host.appendChild(template.content.cloneNode(true));
</script>
```

deep copy

HTML Templates are Document Fragments

```
<html>
  <head></head>
  ▼<body>
    <h1>W3C HTML Template</h1>
    ▶<div id="main">...</div>
    ▼<template id="template">
      ▼#document-fragment
        ▼<div>
          "Hello from "
          <span class="name">Template</span>
        </div>
      </template>
    ▶<script>...</script>
  </body>
</html>
```

DOM

```
<!doctype html>
<h1>W3C HTML Template</h1>
<div id="main"></div>
<template id="template">
  <div>Hello from <span
class="name">Template</span>
  </div>
</template>

<script>
var host = document.querySelector('#main');
var template = document.querySelector('#template');
host.appendChild(template.content.cloneNode(true));
host.appendChild(template.content.cloneNode(true));
host.appendChild(template.content.cloneNode(true));
</script>
```

HTML

HTML Template Praxisbeispiel

Tabellenzeile als Template

für große Tabellen
Performanzgewinne

Vertiefung:

<https://www.html5rocks.com/en/tutorials/webcomponents/template/>

```
<script>
var data = [
  { name: 'Pillar', color: 'Ticked Tabby', sex: 'Female (neutered)', legs: 3 },
  { name: 'Hederal', color: 'Tuxedo', sex: 'Male (neutered)', legs: 4 },
];
</script>
<table>
  <template id="row">
    <tr><td><td><td><td>
  </template>
</table>
<script>
var template = document.querySelector('#row');
for (var i = 0; i < data.length; i += 1) {
  var cat = data[i];
  var clone = template.content.cloneNode(true);
  var cells = clone.querySelectorAll('td');
  cells[0].textContent = cat.name;
  cells[1].textContent = cat.color;
  cells[2].textContent = cat.sex;
  cells[3].textContent = cat.legs;
  template.parentNode.appendChild(clone);
}
</script>
```

<https://www.webcomponents.org/specs#the-html-template-specification>

<https://www.w3.org/TR/html5/scripting-1.html#the-template-element>

Kombination W3C Template & Shadow DOM

```
<!DOCTYPE html>
...<html> == $0
  <head></head>
  ▼<body>
    <h1>Template & Shadow</h1>
    ▼<div id="shadowHost">
      ▶#shadow-root (open)
        <style>*>{ color: red; }</style>
        ▶<div>...</div>
      </div>
    ▼<template id="tmpl">
      #document-fragment
    </template>
    ▶<script>...</script>
  </body>
</html>
```

Template & Shadow

Hello from Shadow DOM

```
<!doctype html>
<h1>Template &amp; Shadow</h1>
<div id="shadowHost"></div>
<template id="tmpl">
  <style>*>{ color: red; }</style>
  <div>Hello from <span class="name">Shadow DOM</span></div>
</template>

<script>
var shadowHost = document.querySelector('#shadowHost');
var shadowRoot = shadowHost.createShadowRoot();
shadowRoot.appendChild(document.querySelector('#tmpl').content);
</script>
```

Kombination Template & Slots

Deklaration der Slots im Template:

```
1 <template id="element-details-template">
2   <style>
3     details {font-family: "Open Sans Light",Helvetica,Arial}
4     .name {font-weight: bold; color: #217ac0; font-size: 120%}
5     h4 { margin: 10px 0 -8px 0; }
6     h4 span { background: #217ac0; padding: 2px 6px 2px 6px }
7     h4 span { border: 1px solid #cee9f9; border-radius: 4px }
8     h4 span { color: white }
9     .attributes { margin-left: 22px; font-size: 90% }
10    .attributes p { margin-left: 16px; font-style: italic }
11  </style>
12  <details>
13    <summary>
14      <span>
15        <code class="name">&lt;slot name="element-name">NEED NAME</slot>&gt;</code>
16        <i class="desc"><slot name="description">NEED DESCRIPTION</slot></i>
17      </span>
18    </summary>
19    <div class="attributes">
20      <h4><span>Attributes</span></h4>
21      <slot name="attributes"><p>None</p></slot>
22    </div>
23  </details>
24  <hr>
25 </template>
```



Nutzung der Slots:

```
1 <element-details>
2   <span slot="element-name">slot</span>
3   <span slot="description">A placeholder inside a web
4     component that users can fill with their own markup,
5     with the effect of composing different DOM trees
6     together.</span>
7   <dl slot="attributes">
8     <dt>name</dt>
9     <dd>The name of the slot.</dd>
10    </dl>
11  </element-details>
12
13 <element-details>
14   <span slot="element-name">template</span>
15   <span slot="description">A mechanism for holding client-
16     side content that is not to be rendered when a page is
17     loaded but may subsequently be instantiated during
18     runtime using JavaScript.</span>
19 </element-details>
```

HTML oder JavaScript? Deklarativ oder Imperativ?

- Imperativ geht immer: in JavaScript kann man alles programmieren
- aber vieles geht auch deklarativ ⇒ HTML, Templates
- Allgemeines Ziel: Benutzung so deklarativ wie möglich
& Komplexität der Implementierung möglichst verbergen
- möglichst weitreichende Kombinationsmöglichkeiten erlauben
- ⇒ Zielkonflikt: Je weitreichender, desto mehr geht es wieder in Richtung Programmierung
 - Beispiel: BPML erfindet Programmierung auf höherer Ebene neu

Parametrisierungsmöglichkeiten für Web-Komponenten

- HTML-Attribute <custom-element name="xyz" class="a" id="b" width="123">
 - Custom-Attribute: Im Shadow-DOM: this.getAttribute('name')
 - Standard-Attribute this.class, this.id und this.width
- Properties und Setter des CustomElement-Objektes
 - element.property = { key: value, array: [a, b, c] }
- Slots
 - benannt <slot name="aaa">
 - unbenannt <slot>
- EventHandler + CustomEvents für Parametrisierung und Kom. nutzen
- CSS Custom Properties als Parameter nutzen
- Light DOM als Parameter nutzen
 - Im Shadow-DOM: this.innerHTML => Komponente kann eigenen Interpreter besitzen
- Globale Funktionen, Werte, Objekte, ...
 - myFunction als offizielle Schnittstelle einer Komponente
 - Aufruf: if (window.myFunction) myFunction(x);

software-technisch
unschön

Gliederung

1. JavaScript-Module (ES6 module)
2. Web-Komponenten
3. **lit-html**
4. LitElement
5. Marktplätze für Komponenten

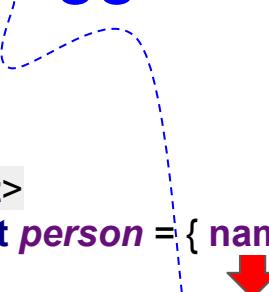
ES6 Template Strings with Backticks `...

```
// Basic literal string creation with ES6 Backtick strings
const s = `In JavaScript '\n' is a line-feed.`  
          ↑  
          ↑  
  
// Multiline ES6 Backtick strings
const t = `In JavaScript this is  
not legal.`  
  
// ES6 Backtick string interpolation
const name = "Bob", time = "today";
console.log(`Hello ${name}, how are you ${time}?`);  
  
// even with expressions to be evaluated
console.log(`The time is ${new Date().toLocaleTimeString()}.`);
```

ES6 Tagged Template Strings with Backticks

```
<script>
  const person = { name: 'Mike', age: 28 };
  console.log( myTag`The person ${person} is a ${person}.` );
    // The person Mike is a youngster.

  function myTag( strings, person ){
    console.log( strings ); // ["The person ", " is a ", "."]
    return strings.slice(0,-1) // Kopie bis zum vorletzten Element
      .join( person.name ) + // Array nach String
      ( person.age > 30 ? 'oldie' : 'youngster' ) +
      strings.slice(-1); // letztes String-Fragment
  }
</script>
```



3kB Templating Library "lit-html"



Transformation von
CommonJS **require** nach
ES6 **import**

```
// inside <script type="module">
import {html, render} from 'https://unpkg.com/lit-html?module';

// This is a lit-html template function. It returns a lit-html template.
const helloTemplate = (name) => html`<div>Hello ${name}!</div>`;

// Call the function with some data, and pass the result to render()

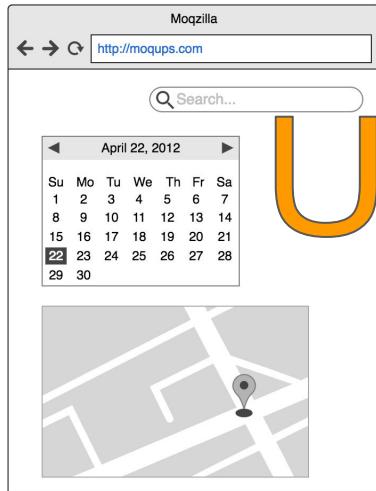
// This renders <div>Hello Steve!</div> to the document body
render(helloTemplate('Steve'), document.body);

// This updates to <div>Hello Kevin!</div>, but only updates the ${name} part
render(helloTemplate('Kevin'), document.body);
```

lit-html

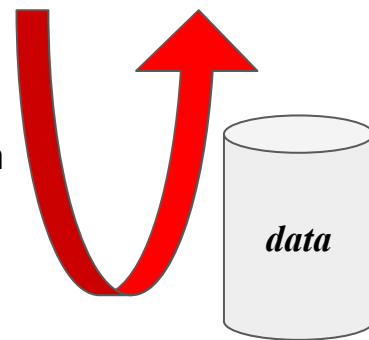
```
let ui = (data) => html`...${data}...`;
```

Funktionales Paradigma



UI = $f(\text{data})$

re-rendering when
data change



- avoid caching intermediate state,
- or doing manual DOM manipulation

Vorteile von lit-html

Efficient

lit-html is **extremely fast**. It uses fast platform features like **HTML <template>** elements with native cloning.

Unlike VDOM libraries, lit-html only ever updates the parts of templates that actually change - it **doesn't re-render the entire view**.

Expressive

lit-html gives you the full power of JavaScript and **functional programming** patterns.

Templates are values that can be computed, passed to and from functions and nested. Expressions are real JavaScript and can include anything you need.

lit-html supports many kind of values natively: strings, DOM nodes, heterogeneous lists, nested templates and more.

Extensible

lit-html is extremely customizable and extensible. Directives customize how values are handled, allowing for asynchronous values, efficient keyed-repeats, error boundaries, and more. lit-html is like your very own **template construction kit**.

Efficiently creating and updating DOM

lit-html is **not a framework**, nor does it include a component model. It focuses on one thing and one thing only: efficiently creating and updating DOM. It can be used **standalone** for simple tasks, or **combined with a framework or component model, like Web Components**, for a full-featured UI development platform.

Efficient Re-Renders

```
<body>
<h1 id="header"></h1>
<button id="Mike">Mike</button>
<button id="Fred">Fred</button>

<script type="module">
import { html, render } from "https://unpkg.com/lit-html?module";
const personTemplate = ( name ) => html`Diary of ${name}`;
const header = document.getElementById('header');
let clickedPerson = null;

['Mike', 'Fred'].forEach( person => {
  const button = document.getElementById( person );
  button.addEventListener( 'click', e => {
    clickedPerson = person;
    render( personTemplate( clickedPerson ), header );
  });
});

</script>
```

Diary of Fred

Mike Fred

Elements Console Sources

```
<!doctype html>
...<html lang="en"> == $0
```

```
► <head>...</head>
```

```
▼ <body>
```

```
  ▼ <h1 id="header">
```

```
    <!-->
```

```
    "Diary of "
```

```
    "Mike"   
```

```
    <!-->
```

```
    <!-->
```

```
</h1>
```

```
<button id="Mike">Mike</button>
```

```
<button id="Fred">Fred</button>
```

```
► <script type="module">...</script>
```

```
</body>
```

```
</html> https://lit-html.polymer-project.org/
```

When you call render, lit-html only updates the parts of the template that have changed since the last render. This makes lit-html updates very fast.

lit-html: types of bindings mit " ? . @ "

- Text:

```
html`<h1>Hello ${name}</h1>`
```

- Property:

```
html`<input .value=${value}>`
```

- Attribute:

```
html`<div id=${id}></div>`
```

- Event Listener:

```
html`<button @click=${(e) => console.log('clicked')}>Click Me</button>`
```

- Boolean Attribute:

```
html`<input ?disabled=${disabled}>`
```

<https://polymer.github.io/lit-html/guide/writing-templates>

lit-html: Ternary Operator

```
html`  
  ${user.isLoggedIn}  
    ? html`Welcome ${user.name}`  
    : html`Please log in`  
  }  
`;
```

<https://polymer.github.io/lit-html/guide/writing-templates>

lit-html: Nesting Templates in Template

```
const header = html`<h1>Header</h1>`;
```

```
const page = html`  
  ${header}  
  <p>This is some text</p>  
`;
```

lit-html: DOM nodes in Templates

```
const div = document.createElement('div');
const page = html`  
  ${div}  
  <p>This is some text</p>  
`;
```

<https://polymer.github.io/lit-html/guide/writing-templates>

lit-html: Looping

```
html`  
  <ul>  
    ${items.map((i) => html`<li>${i}</li>`)}  
  </ul>  
`;
```

<https://polymer.github.io/lit-html/guide/writing-templates>

lit-html: Looping -2-

```
const itemTemplates = [];
for (const i of items) {
  itemTemplates.push(html`<li>${i}</li>`);
}

html`
<ul>
  ${itemTemplates}
</ul>
`;
```

<https://polymer.github.io/lit-html/guide/writing-templates>

Awesome lit-html

<https://github.com/web-padawan/awesome-lit-html>

A curated list of awesome lit-html resources.

lit-html — an efficient, expressive, extensible HTML templating library for JavaScript.

Contents

- [General resources](#)
- [Community](#)
- [Overview](#)
- [Implementations](#)
 - [Renderers](#)
 - [LitElement Extensions](#)
 - [Other](#)

- [Components](#)
 - [Component Libraries](#)
 - [Individual Components](#)
- [Starter Templates](#)
- [IDE Plugins](#)
- [TypeScript Plugins](#)
- [Tools](#)
- [Static Site Generators](#)
- [Examples](#)
- [Videos](#)
- [Podcasts](#)
- [Blogs](#)
- [Projects](#)
- [Inspired Solutions](#)
- [Other awesome resources](#)

<https://github.com/web-padawan/awesome-lit-html>

VS Code Plugin for lit-html



lit-html

Matt Bierner | ↓ 64,701 installs | ★★★★★ (10) | Free

Syntax highlighting and IntelliSense for html inside of JavaScript and TypeScript tagged template strings

```
const buttonTemplate = (title = 'submit') => html`  
  <button>  
    ${title}  
  </button>  
  
render(buttonTemplate(), document.body)
```

<https://marketplace.visualstudio.com/items?itemName=bierner.lit-html>

Gliederung

1. JavaScript-Module (ES6 module)
2. Web-Komponenten
3. lit-html
4. LitElement
5. Marktplätze für Komponenten

LitElement

- A simple base class for creating fast, lightweight web components
- LitElement makes it easy to define Web Components – ideal for sharing elements across your organization or building a UI design system.
- Use your components anywhere you use HTML: in your main document, a CMS, Markdown, or a framework like React or Vue.

Delightfully declarative

LitElement's simple, familiar development model makes it easier than ever to build Web Components.

Express your UI declaratively, as a function of state. No need to learn a custom templating language – you can use the full power of JavaScript in your templates. Elements update automatically when their properties change.

Fast and light

Whether your end users are in emerging markets or Silicon Valley, they'll appreciate that LitElement is extremely fast.

LitElement uses [lit-html](#) to define and render HTML templates. DOM updates are lightning-fast, because lit-html only re-renders the dynamic parts of your UI – no diffing required.

Seamlessly interoperable

LitElement follows the [Web Components standards](#), so your components will work with any framework.

LitElement uses Custom Elements for easy inclusion in web pages, and Shadow DOM for encapsulation. There's no new abstraction on top of the web platform.

LitElement

- Attribut-Werte werden immer automatisch in Properties konvertiert.
- Property-Werte werden nur in Attributen gespiegelt, falls `reflect: true`

```
<script type="module">
import { LitElement, html } from 'https://unpkg.com/lit-element/lit-element.js?module';

class MyElement extends LitElement {
  static properties = { // Welche Props soll LitElement verwalten?
    prop1: { type: String },
    prop2: { type: Number, reflect: true } // if attribute should reflect property changes
  };
  constructor() {
    super();
    this.prop1 = '';
    this.prop2 = 0;
  }
  render() {
    return html`

prop1 ${this.prop1}</p>
    <p>prop2 ${this.prop2}</p>
    <button @click="${this.changeProperties}">change properties</button>
    <button @click="${this.changeAttributes}">change attributes</button>
  `;
  }
}


```

Declared Properties

Initialisierung

re-render if declared property
or attribute changes

```
changeAttributes() {
  let randy = Math.floor(Math.random()*10);
  this.setAttribute('prop1', randy.toString());
  this.setAttribute('prop2', randy.toString());
  this.requestUpdate();
}

changeProperties() {
  let randy = Math.floor(Math.random()*10);
  this.prop1 = randy.toString();
  this.prop2 = randy;
}

customElements.define('my-element', MyElement);
</script>
<my-element></my-element>
```

<https://lit-element.polymer-project.org/guide/properties#conversion>

LitElement Garantien

- Re-Render bei jeder Property Change
 - nur in static properties deklarierte Properties
 - minimales Re-Rendering: nur die Teile im DOM, die sich wirklich geändert haben
- Attribut-Konvertierung typkonform
 - Attribute: String ⇒ Property: String, Number, Boolean, Object, Array.
- Property ⇒ Attribut
 - `toString()`

The following options are available:

- `type`: Use LitElement's default attribute converter.
- `converter`: Convert between properties and attributes.
- `attribute`: Configure observed attributes.
- `reflect`: Configure reflected attributes.
- `noAccessor`: Whether to set up a default property accessor.
- `hasChanged`: Specify what constitutes a property change.

LitElement - Tutorial

<https://dev.to/link2twenty/litelement-using-web-components-580a>

LitElement Styles

```
import { LitElement, html, css } from 'lit-element';

const mainColor = css`red`;

class MyElement extends LitElement {
  static get styles() {
    return css`
      div { color: ${mainColor} }
    `;
  }
  render() {
    return html`<div>Some content in a div</div>`;
  }
}

customElements.define('my-element', MyElement);
```

aus
Sicherheitsgründen
nur CSS Templates

Gliederung

1. JavaScript-Module (ES6 module)
2. Web-Komponenten
3. lit-html
4. LitElement
5. Marktplätze für Komponenten

Komponentenmarkt für W3C-Web-Komponenten

The screenshot shows the homepage of [WebComponents.org](https://www.webcomponents.org/). At the top, there's a logo for **WEBCOMPONENTS.ORG** and a blue button labeled **Publish element**. Below that is a search bar with the placeholder **Search custom elements**. To the right of the search bar are three categories: **NOTIFICATION**, **MEDIA**, and **IMAGE**. A large yellow callout box on the right side contains the text **über 2000 Komponenten**. On the left, under the heading **Featured elements**, there are three cards:

- poly-prep**: Helper elements to load polymer elements. Author: **bahrus**.
- service-weezev...**: web component tools for api weeevent. Author: **Raulnet**.
- tp-person**: tp-person. Author: **avizcaino**.

A yellow arrow points from the text "über 2000 Komponenten" towards the "Browse elements" link at the top right of the page.

<https://www.webcomponents.org/>



Digital Makerspace: App = Komponente + Konfig

Digital Makerspace | Home Apps Components Publish

Deutsch English Log In

The screenshot shows a grid of 18 app components:

- Achievements** by Artur Zimmermann (v1.0.0): Enables the Gamification Elements of collecting Achievement. Rating: ★★★★☆ (0).
- Analog Clock** by Manfred Kaul (v3.0.1): Rating: ★★★★★ (0).
- App Collection** by Tea Kless (v1.0.0): Combine your apps into an app collection. Rating: ★★★★★ (0).
- Badges** by Artur Zimmermann (v1.0.0): Gamification Element to collect Badges. Rating: ★★★★★ (0).
- Cloze Builder** by André Kless (v3.0.3): Configuration Builder for the 'Fill in the Blank' component. Rating: ★★★★★ (0).
- Comments** by André Kless (v4.1.0): Create a discussion with rateable comments. Rating: ★★★★★ (0).
- Comparing Games** by Artur Zimmermann (v1.0.0): Compare games with other User. Rating: ★★★★★ (0).
- Content** by André Kless (v5.4.3): Create content and use it anywhere on the web. Rating: ★★★★★ (1).
- Data Storage** by André Kless (v1.0.1): Create an app that lets you manage the data in a data store. Rating: ★★★★★ (1).
- Exercise** by André Kless (v5.0.0): Create a free text task. Rating: ★★★★★ (0).
- Fill in the Blank** by André Kless (v6.0.1): Create a task with missing words in a text. Rating: ★★★★★ (1).
- Guess Picture** by André Kless (v1.1.0): Create your own image guessing game. Rating: ★★★★★ (1).
- Highchart** by André Kless (v3.0.1): Create your own Highchart. Rating: ★★★★★ (1).
- JSON Builder** by André Kless (v1.4.2): Create data in JavaScript Object Notation (JSON). Rating: ★★★★★ (1).
- Kanban Board** by André Kless (v2.0.2): Create a kanban board and use it in your context. Rating: ★★★★★ (1).
- Kanban Card** by André Kless (v2.0.2): Create kanban cards for your task organization. Rating: ★★★★★ (1).
- Listing** by André Kless (v3.3.0): Create a freely definable listing. Rating: ★★★★★ (1).
- Live Poll** by André Kless (v2.3.2): Create a spontaneous live survey. Rating: ★★★★★ (1).

Berlin (selected), New York, Large Berlin, Create Similar App

Demos

Create Similar App

67

<https://ccmjs.github.io/digital-maker-space/>

"frameworkless"

Marktplätze für Web-Komponenten

- [Amber Components](#) - Web Components implementation of the Amber Design System.
- [Blackstone UI](#) - Web components for creating interfaces built with lit-html and LitElement.
- [Bronconents](#) - Modern Web Components built with Lit-Element.
- [Carbon Custom Elements](#) - Experimental variant of Carbon Design System built with Web Components.
- [Chartjs Web Components](#) - Web components for chartjs.
- [Clever components](#) - Collection of Web Components made by Clever Cloud.
- [Ink Components](#) - Web components for interactive scientific writing.
- [LRNWebComponents](#) - ELMS:LN produced web components for any project.
- [Material Web Components](#) - Material Design implemented as Web Components.
- [Microsoft Graph Toolkit](#) - Collection of web components for the Microsoft Graph.
- [UI5 Web Components](#) - Enterprise-flavored sugar on top of native APIs!
- [VSCode-Webview-Elements](#) - Components for creating VSCode extensions which use the Webview API.
- [Weightless](#) - High quality web components with a small footprint.
- [Wired Elements](#) - Collection of elements that appear hand drawn.
- <https://developer.salesforce.com/docs/component-library/overview/components>
- <https://custom-elements-everywhere.com>
- <https://getmdl.io/>
- <https://bit.dev> also with frameworks like React, Angular, Vue, ...



Zusammenfassung

1. JavaScript-Module (ES6 module)
2. W3C Web-Komponenten
3. lit-html für Tagged Template Strings
4. LitElement als Superklasse von Web-Komponenten
5. Marktplätze für universell kombinierbare Web-Komponenten

Use
The
Platform



WEBCOMPONENTS.ORG

Building blocks for the web