Web Components

Web Components is a suite of different technologies allowing you to create reusable custom elements — with their functionality encapsulated away from the rest of your code — and utilize them in your web apps.

Concepts and usage

As developers, we all know that reusing code as much as possible is a good idea. This has traditionally not been so easy for custom markup structures — think of the complex HTML (and associated style and script) you've sometimes had to write to render custom UI controls, and how using them multiple times can turn your page into a mess if you are not careful.

Web Components aims to solve such problems — it consists of three main technologies, which can be used together to create versatile custom elements with encapsulated functionality that can be reused wherever you like without fear of code collisions.

- **Custom elements**: A set of JavaScript APIs that allow you to define custom elements and their behavior, which can then be used as desired in your user interface.
- **Shadow DOM**: A set of JavaScript APIs for attaching an encapsulated "shadow" DOM tree to an element — which is rendered separately from the main document DOM — and controlling associated functionality. In this way, you can keep an element's features private, so they can be scripted and styled without the fear of collision with other parts of the document.
- **HTML templates**: The `<template>` and `<slot>` elements enable you to write markup templates that are not displayed in the rendered page. These can then be reused multiple times as the basis of a custom element's structure.

The basic approach for implementing a web component generally looks something like this:

1. Create a class in which you specify your web component functionality, using the ECMAScript 2015 class syntax (see Classes for more information).
2. Register your new custom element using the `CustomElementRegistry.define()` method, passing it the element name to be defined, the class or function in which its functionality is specified, and optionally, what element it inherits from.
3. If required, attach a shadow DOM to the custom element using `Element.attachShadow()` method. Add child elements, event listeners, etc., to the shadow DOM using regular DOM methods.
4. If required, define an HTML template using `<template>` and `<slot>`. Again use regular DOM

methods to clone the template and attach it to your shadow DOM.
5. Use your custom element wherever you like on your page, just like you would any regular HTML element.

Tutorials

Using custom elements

A guide showing how to use the features of custom elements to create simple web components, as well as looking into lifecycle callbacks and some other more advanced features.

Using shadow DOM

A guide that looks at shadow DOM fundamentals, showing how to attach a shadow DOM to an element, add to the shadow DOM tree, style it, and more.

Using templates and slots

A guide showing how to define a reusable HTML structure using `<template>` and `<slot>` elements, and then use that structure inside your web components.

Reference

Custom elements

`CustomElementRegistry`

Contains functionality related to custom elements, most notably the `CustomElementRegistry.define()` method used to register new custom elements so they can then be used in your document.

`Window.customElements`

Returns a reference to the `CustomElementRegistry` object.

Life cycle callbacks

Special callback functions defined inside the custom element's class definition, which affect its behavior:

- `connectedCallback`: Invoked when the custom element is first connected to the document's DOM.
- `disconnectedCallback`: Invoked when the custom element is disconnected from the document's DOM.
- `adoptedCallback`: Invoked when the custom element is moved to a new document.
- `attributeChangedCallback`: Invoked when one of the custom element's attributes is added, removed, or changed.

Extensions for creating custom built-in elements

- The `is` global HTML attribute: Allows you to specify that a standard HTML element should behave like a registered custom built-in element.
- The "is" option of the `Document.createElement()` method: Allows you to create an instance of a standard HTML element that behaves like a given registered custom built-in element.

CSS pseudo-classes

Pseudo-classes relating specifically to custom elements:

- `:defined`: Matches any element that is defined, including built in elements and custom elements defined with `CustomElementRegistry.define()`).
- `:host`: Selects the shadow host of the shadow DOM containing the CSS it is used inside.
- `:host()`: Selects the shadow host of the shadow DOM containing the CSS it is used inside (so you can select a custom element from inside its shadow DOM) — but only if the selector given as the function's parameter matches the shadow host.
- `:host-context()`: Selects the shadow host of the shadow DOM containing the CSS it is used inside (so you can select a custom element from inside its shadow DOM) — but only if the selector given as the function's parameter matches the shadow host's ancestor(s) in the place it sits inside the DOM hierarchy.

CSS pseudo-elements

Pseudo-elements relating specifically to custom elements:

- `::part`: Represents any element within a shadow tree that has a matching `part` attribute.

Shadow DOM

`ShadowRoot`

Represents the root node of a shadow DOM subtree.

`DocumentOrShadowRoot`

A mixin defining features that are available across document and shadow roots.

`Element` extensions

Extensions to the `Element` interface related to shadow DOM:

- The `Element.attachShadow()` method attaches a shadow DOM tree to the specified element.

- The `Element.shadowRoot` property returns the shadow root attached to the specified element, or `null` if there is no shadow root attached.

Relevant `Node` additions

Additions to the `Node` interface relevant to shadow DOM:

- The `Node.getRootNode()` method returns the context object's root, which optionally includes the shadow root if it is available.
- The `Node.isConnected` property returns a boolean indicating whether or not the Node is connected (directly or indirectly) to the context object, e.g. the `Document` object in the case of the normal DOM, or the `ShadowRoot` in the case of a shadow DOM.

`Event` extensions

Extensions to the `Event` interface related to shadow DOM:

- `Event.composed`: Returns a `Boolean` which indicates whether the event will propagate across the shadow DOM boundary into the standard DOM (`true`), or not (`false`).
- `Event.composedPath`: Returns the event's path (objects on which listeners will be invoked). This does not include nodes in shadow trees if the shadow root was created with `ShadowRoot.mode` closed.

HTML templates

`<template>`

Contains an HTML fragment that is not rendered when a containing document is initially loaded, but can be displayed at runtime using JavaScript, mainly used as the basis of custom element structures. The associated DOM interface is `HTMLTemplateElement`.

`<slot>`

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. The associated DOM interface is `HTMLSlotElement`.

The `slot` global HTML attribute

Assigns a slot in a shadow DOM shadow tree to an element.

`Slotable`

A mixin implemented by both `Element` and `Text` nodes, defining features that allow them to become the contents of an `<slot>` element. The mixin defines one attribute, `Slotable.assignedSlot`, which returns a reference to the slot the node is inserted in.

`Element` extensions

Extensions to the `Element` interface related to slots:

- `Element.slot`: Returns the name of the shadow DOM slot attached to the element.

CSS pseudo-elements

Pseudo-elements relating specifically to slots:

- `::slotted`: Matches any content that is inserted into a slot.

The `slotchange` event

Fired on an `HTMLSlotElement` instance (`<slot>` element) when the node(s) contained in that slot change.

Examples

We are building up a number of examples in our web-components-examples GitHub repo. More will be added as time goes on.

Specifications

Specification	Status	Comment
HTML Living Standard The definition of '<template> element' in that specification.	LS Living Standard	The definition of `<template>`.
HTML Living Standard The definition of 'custom elements' in that specification.	LS Living Standard	The definition of HTML Custom Elements.
DOM The definition of 'shadow trees' in that specification.	LS Living Standard	The definition of Shadow DOM.
HTML Imports	WD Working Draft	Initial HTML Imports definition.
Shadow DOM	O Obsolete	Initial Shadow DOM definition.

Browser compatibility

In general:

- Web components are supported by default in Firefox (version 63), Chrome, and Opera.
- Safari supports a number of web component features, but less than the above browsers.
- Edge is working on an implementation.

For detailed browser support of specific features, you'll have to consult the reference pages listed above.

See also

- Open Web Components — Guides, tools and libraries for developing web components.
- DataFormsJS — Open source web components library — Set of Web Components that can be used to build Single Page Apps (SPA), Display JSON data from API's and Web Services, and bind data to different elements on screen. All Web Components are plain JavaScript and require no build process.
- FAST is a web component library built by Microsoft which offers several packages to leverage depending on your project needs. Fast Element is a lightweight means to easily build performant, memory-efficient, standards-compliant Web Components. Fast Foundation is a library of Web Component classes, templates, and other utilities built on fast-element intended to be composed into registered Web Components.
- Hybrids — Open source web components library, which favors plain objects and pure functions over `class` and `this` syntax. It provides a simple and functional API for creating custom elements.
- Polymer — Google's web components framework — a set of polyfills, enhancements, and examples. Currently the easiest way to use web components cross-browser.
- Snuggsi — Easy Web Components in ~1kB *Including polyfill* — All you need is a browser and basic understanding of HTML, CSS, and JavaScript classes to be productive.
- Slim.js — Open source web components library — a high-performant library for rapid and easy component authoring; extensible and pluggable and cross-framework compatible.
- Stencil — Toolchain for building reusable, scalable design systems in web components.