

# Web Component

---

유성민

2019

+ WEB

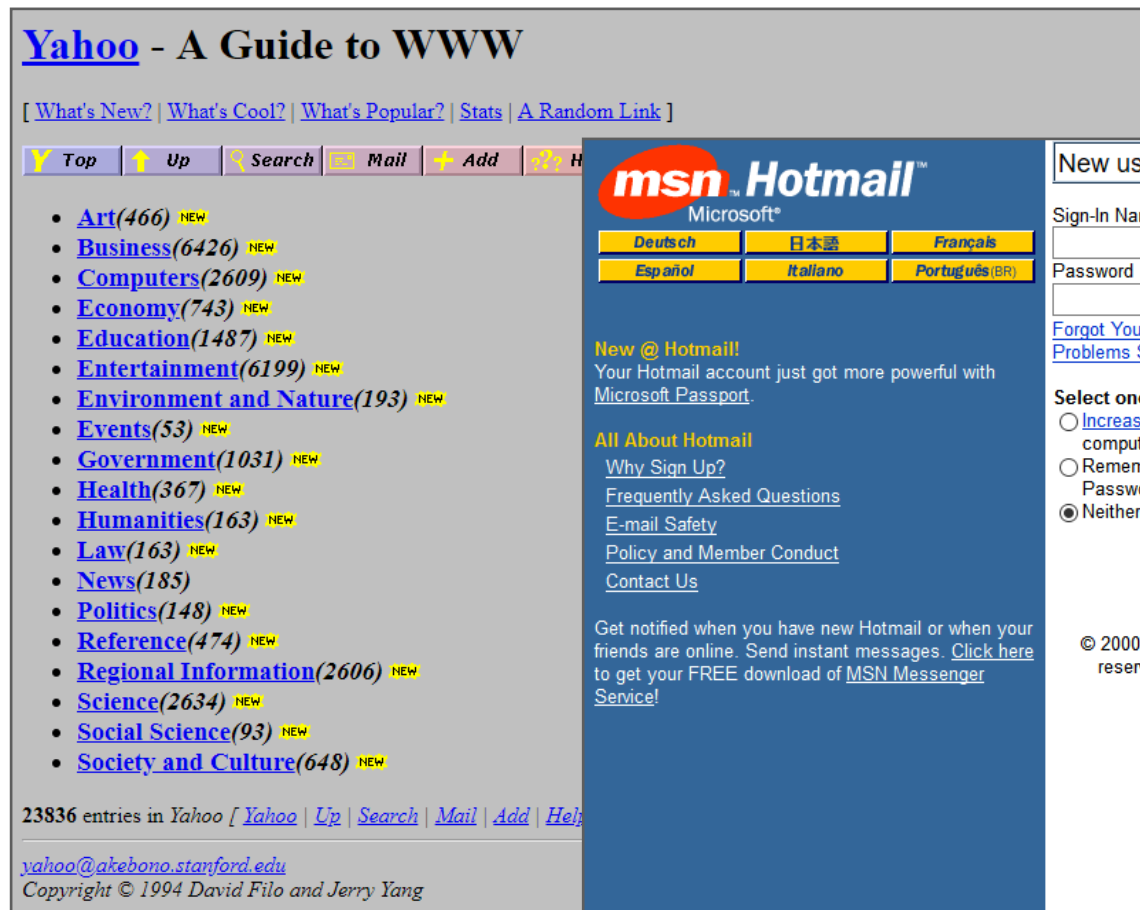
# HTML

HTML은 **하이퍼텍스트 마크업 언어**(HyperText 문서와 문서가 링크로 연결, Markup 정보를 정의하는 태그, Language 언어), 제목, 단락, 목록 등과 같은 본문을 위한 구조적 의미를 나타내는 것뿐만 아니라 링크, 인용과 그 밖의 항목으로 구조적 문서를 만들 수 있는 방법을 제공

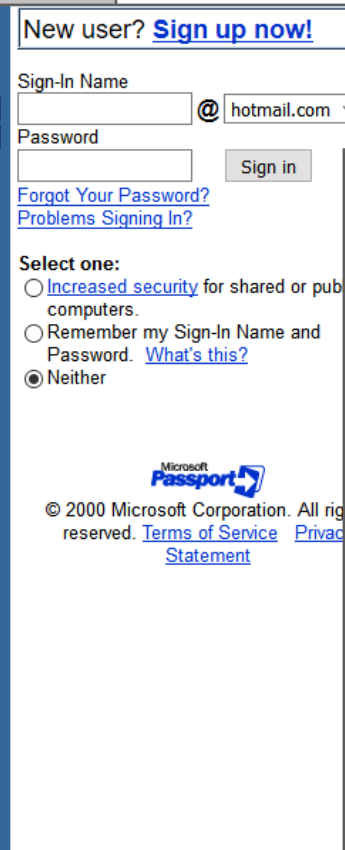
# HTML

## Web Component

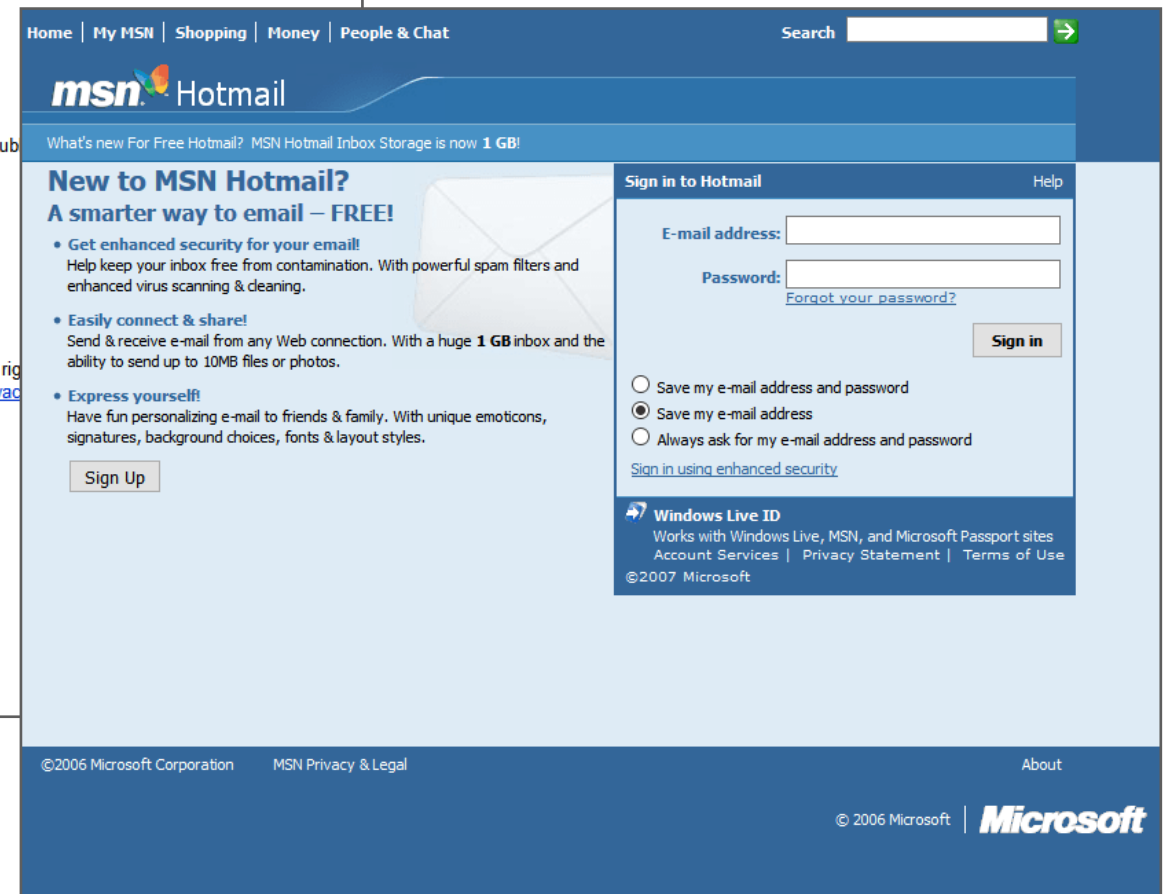
1994년



2000년



2006년



## HyperText Markup Language

웹 페이지를 다른 웹 페이지나 같은 웹 페이지의 특정 부분과 **연결** (하이퍼텍스트) 글, 이미지 등의 다양한 콘텐츠를 **표시** (마크업)

# HTML

## Web Component

### Web API

W3C (HTML5)

WHATWG (HTML Living Standard)

ECMA International (TC39 위원회, JavaScript)

### Web Application

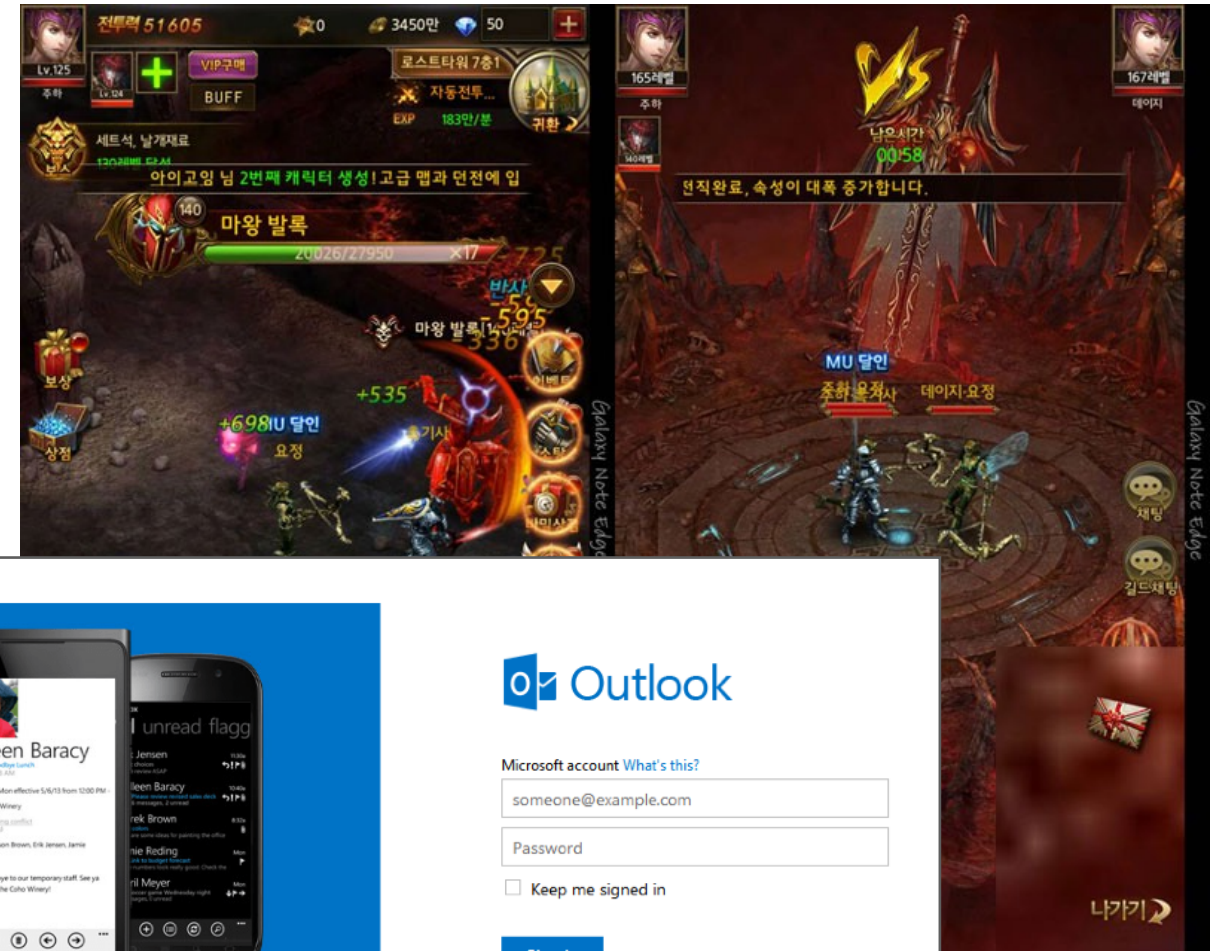
XMLHttpRequest(AJAX), jQuery, SPA, PWA ...

### \* W3C + WHATWG (2019.05.28)

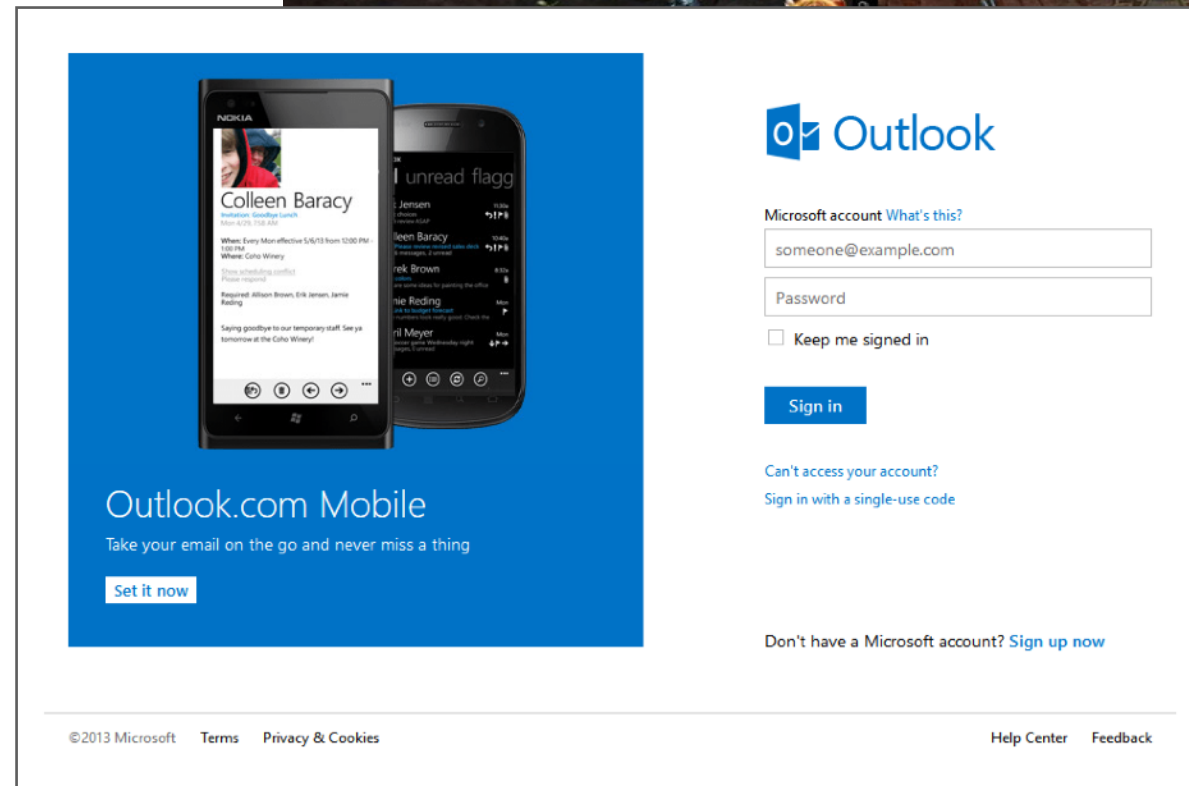
"W3C와 WHATWG가 HTML과 DOM 규격을  
단일 버전으로 개발하는 데 협력하는 합의안에 서명"

단순 HTML 에서 웹 애플리케이션으로 진화

2013년



2018년



[출처] <http://game.donga.com/90308/>

[참고] <http://www.zdnet.co.kr/view/?no=20190531184644>

# HTML

## Web Component

HTML이 단순 웹 문서에서 대규모 어플리케이션(대규모 HTML, 복잡한 UI)화 되면서  
자연스럽게 해결해야할 과제들에 직면. (전통적인 HTML 고려사항)

### 고유 ID 속성값

한 페이지에 중복된 id="속성값" 선언 주의

### 글로벌(global scope) CSS Styles

네이밍, 적용 우선순위(!important),  
트리구조(탐색레벨, 상위 속성 상속) 성능/복잡도,  
외부 코드에 의한 오염 가능성, 미사용 코드 검출 어려움

### 비표준 방식의 HTML 재활용

script 문자열로 처리  
<script type="text/template"></script>  
mustache, handlebars, ejs 등 외부 template 도구  
text 파일 -> 파싱(string, code) -> 컴파일(html) -> 렌더

### 리플로우/리페인트

대규모/복잡한 HTML, 다양한 동적 UI





















페이지 내부 기능/역할단위로 분리된 개발, 코드 재사용에 대한 요구를 어떻게 효율적으로 수용할 것인가

# 라이브러리/프레임워크

## Web Component

Backbone.js, AngularJS, React, Vue 등  
다양한 도구(라이브러리/프레임워크)를 활용  
HTML 모듈화/캡슐화/컴포넌트화 대응

수많은 라이브러리/프레임워크 전쟁터

<b>React / Redux</b>  <b>RealWorld</b> example app Star 3k Fork 857	<b>Angular</b>  <b>RealWorld</b> example app Star 3k Fork 1149	<b>Elm</b>  <b>RealWorld</b> example app Star 2k Fork 319
<b>Vue</b>  <b>RealWorld</b> example app Star 1k Fork 353	<b>React / MobX</b>   <b>RealWorld</b> example app Star 756 Fork 157	<b>AngularJS</b>  <b>RealWorld</b> example app Star 290 Fork 223
<b>PureScript + Halogen</b>  <b>RealWorld</b> example app Star 274 Fork 17	<b>ClojureScript + re-frame</b>  <b>RealWorld</b> example app Star 191 Fork 22	<b>Aurelia</b>  <b>RealWorld</b> example app Star 155 Fork 17
<b>Angular + ngrx + nx</b>   <b>RealWorld</b> example app Star 123 Fork 37	<b>Svelte / Sapper</b>  <b>RealWorld</b> example app Star 112 Fork 9	<b>AppRun</b>  <b>RealWorld</b> example app Star 50 Fork 11
<b>ClojureScript + Keechma</b>  <b>RealWorld</b> example app Star 37 Fork 2	<b>Dojo 2</b>  <b>RealWorld</b> example app Star 20 Fork 3	<b>Hyperapp 1</b>  <b>RealWorld</b> example app Star 21 Fork 2
<b>Stencil.js</b>  <b>RealWorld</b> example app Star 9 Fork 1	<b>Crizmas MVC</b>  <b>RealWorld</b> example app Star 6 Fork 2	<b>Imba</b>  <b>RealWorld</b> example app Star 4 Fork 0

# Angular

## Web Component

```
<!-- app.component.html -->
<h2>{{ name }}</h2>
<p>
  Start editing to see some magic happen :)
</p>

<!-- index.html -->
<!-- Angular 컴포넌트에 의해 -->
<my-app>loading</my-app>

<!-- 이렇게 바뀌어 렌더링 -->
<style>
  /* styles 지역 CSS */
  h2[_ngcontent-vve-c17] {
    color: #673ab7;
  }
</style>
<my-app _nghost-vve-c17="" ng-version="8.0.0">
  <h2 _ngcontent-vve-c17="">Angular</h2>
  <p _ngcontent-vve-c17="">
    Start editing to see some magic happen :)
  </p>
</my-app>
```

```
// app.component.ts - TypeScript
import { Component } from '@angular/core';

@Component({ // @Component 데코레이터
  selector: 'my-app',
  // template
  /*template: `
    <h2>{{ name }}</h2>
    <p>
      Start editing to see some magic happen :)
    </p>
  `,*/
  // 또는
  templateUrl: './app.component.html',
  // css
  styles: [`
    h2 { color: #673ab7; }
  `],
  // 또는
  //styleUrls: [ './app.component.css' ] // 전역
})

export class AppComponent { // 컴포넌트 클래스 선언 및 export
  name = 'Angular';
}
```

[참고] 버전 1.x 기준이 아닌 2.x 이상, 2016년 9월 14일 발표, @Component 데코레이터 사용

[출처] <https://poiemaweb.com/angular-component-basics>



# Vue

## Web Component

```
<!-- Vue 컴포넌트에 의해 -->
<div id="example">
  <my-component></my-component>
</div>
```

```
<!-- 이렇게 바뀌어 렌더링 -->
<div id="example">
  <div>사용자 정의 컴포넌트 입니다!</div>
</div>
```

```
// 전역 등록
Vue.component('my-component', {
  template: '<div>사용자 정의 컴포넌트 입니다!</div>'
})
```

```
// 루트 인스턴스 생성 - 전역 등록된 컴포넌트 사용
new Vue({
  el: '#example'
})
```

```
// 지역 등록
new Vue({
  el: '#example',
  components: {
    // <my-component> 는 상위 템플릿에서만 사용할 수 있습니다.
    'my-component': {
      template: '<div>사용자 정의 컴포넌트 입니다!</div>'
    }
  }
  // ...
})
```

[참고] 2014년 2월 발표

[출처] <https://kr.vuejs.org/v2/guide/components.html>



# React

## Web Component

```
// hello.js
import React from 'react';
export default ({ name }) => <h1>Hello {name}!</h1>;

<!-- React 컴포넌트에 의해 -->
<div id="root"></div>

<!-- 이렇게 바뀌어 렌더링 -->
<div id="root">
  <div>
    <h1>Hello React!</h1>
    <p style="background-color: red;">
      Start editing to see some magic happen :)
    </p>
  </div>
</div>
```

```
// index.js - JSX
import React, { Component } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';
import './style.css'; // 전역

class App extends Component {
  constructor() {
    super();
  }
  render() {
    const style = { // 지역
      backgroundColor : 'red',
    }
    return (
      <div>
        <Hello name={this.props.name} />
        <p style={style}>
          Start editing to see some magic happen :)
        </p>
      </div>
    );
  }
}

render(<App name="React" />, document.getElementById('root'));
```

# 웹컴포넌트

## Web Component

컴포넌트 단위 구현을 위한, 웹 표준 API 집합

Custom Elements

HTML Template Element  
(JavaScript Template literals)

lit-html

Shadow DOM

ES Modules  
(HTML Imports)

...

[참고] <https://github.com/w3c/webcomponents/>

[참고] <https://fronteers.nl/congres/2011/sessions/web-components-and-model-driven-views-alex-russell>

# HTML Template Element

## Web Component

페이지가 로딩되는 동안 파서가 <template> element 콘텐츠를 처리하지만, 콘텐츠가 유효한지만 확인합니다.

<template> element 콘텐츠는 렌더링되지 않습니다.

HTML, JavaScript 역할 분리, 재사용

```
<!-- html //-->
<table></table>

<!-- template element //-->
<template id="template">
  <tr>
    <td class="title">
      <button value=""></button>
    </td>
    <td class="date"></td>
  </tr>
</template>
```

```
if('content' in document.createElement('template')) {
  let template = document.querySelector('#template');
  let clone = template.content.cloneNode(true);
  let button = clone.querySelector('button');
  let date = clone.querySelector('.date');

  button.setAttribute('value', 'test');
  button.textContent = 'test';
  date.textContent = 'date';

  document.querySelector('table').appendChild(document.importNode(clone));
}
```

# JavaScript Template Literals

## Web Component

기존 문자열 결합 사용형태와 비교

```
var name1 = 'ysm', name2 = '유성민';
var number1 = 1, number2 = 1;

// 기존 문자열 결합
['template test "', name1, '"', '"', name2, '" !!'].join(''); // template test "ysm", "유성민" !!
'number test ' + number1 + '+' + number2 + '=' + (number1+number2); // number test 1+1=2

// ES6 Template Literals
// ` 백틱(backtick) 문자, ${} 사용
`template test "${name1}", "${name2}" !!`; // template test "ysm", "유성민" !!
`number test ${number1}+${number2}=${number1+number2}`; // number test 1+1=2
```

# JavaScript Template Literals

## Web Component

기존 문자열 결합 사용형태와 비교

```
var test = 'test';
var template1 = '<div>' + test + '</div>';
var template2 = '\
  <div>\
    ' + test + '\
  </div>';
var template3 = [
  '<div>',
  test,
  '</div>'
].join('');
```

```
let test = 'test';
let templateLiterals = `
  <div>
    ${test}
  </div>
`;
```

# Custom Elements

## Web Component

"HTML은 사용하기가 까다롭지 않고 유연합니다."

예를 들어, 페이지에 `<ysm></ysm>`를 선언하면 브라우저가 이를 완전히 수락합니다.

비표준 태그가 작동하는 이유는 HTML 사양이 이를 허용하기 때문입니다.

사양에 정의되지 않은 요소는 `HTMLUnknownElement`로 파싱됩니다.

### 사용자정의 요소 생성 관련 규칙

1. 사용자정의 요소의 이름에는 **대시(-)**가 포함되어야 합니다.

`<x-tags>`, `<my-element>` 및 `<my-awesome-app>`은 모두 유효한 이름이지만, `<tabs>` 및 `<foo_bar>`는 그렇지 않습니다.

이러한 요구사항은 HTML 파서가 일반 요소와 사용자설정 요소를 구별할 수 있도록 합니다.

또한 새로운 태그가 HTML에 추가될 때 다음 버전과의 호환성도 보장되도록 합니다.

2. 동일한 태그를 **두 번 이상 정의(요소확장/요소업그레이드)**할 수 없습니다.

중복 정의 시 `DOMException`이 발생합니다.

새로운 태그(사용자 요소)에 대해 브라우저에 알리고 나면 그걸로 끝입니다. 취소할 수 없습니다.

3. HTML은 몇 가지 요소만 스스로 닫도록 허용하므로 **사용자설정 요소는 스스로 닫을 수 없습니다.**

항상 닫는 태그를 작성해야 합니다. (예를 들어 `<app-drawer></app-drawer>`)

[HTML 사양] <https://html.spec.whatwg.org/multipage/dom.html#htmlunknownelement>

[출처] <https://developers.google.com/web/fundamentals/web-components/customelements?hl=ko>

# Custom Elements

## Web Component

사용자 요소 정의 (요소 확장/요소 업그레이드)

```
// [일반적인 요소 생성] createElement
var ysmTest = document.createElement("ysm-test");
ysmTest.innerText = "유성민";
document.body.appendChild(ysmTest);
```

```
<!-- body html -->
<body>
  <ysm-test>유성민</ysm-test>
</body>
```

```
// [요소 확장/업그레이드] class
class TestYSM extends HTMLElement {
  constructor() {
    // 항상 생성자에서 super는 처음으로 호출됩니다
    super();

    // 엘리먼트의 기능들은 여기에 작성합니다.
    // ...
  }
}
```

```
// define (요소정의)
customElements.define('test-ysm', TestYSM);

<!-- custom element html -->
<test-ysm></test-ysm>
```

[lifecycle callbacks] <https://developers.google.com/web/fundamentals/web-components/customelements>

[ECMAScript 6, JavaScript class] <https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Classes>



```

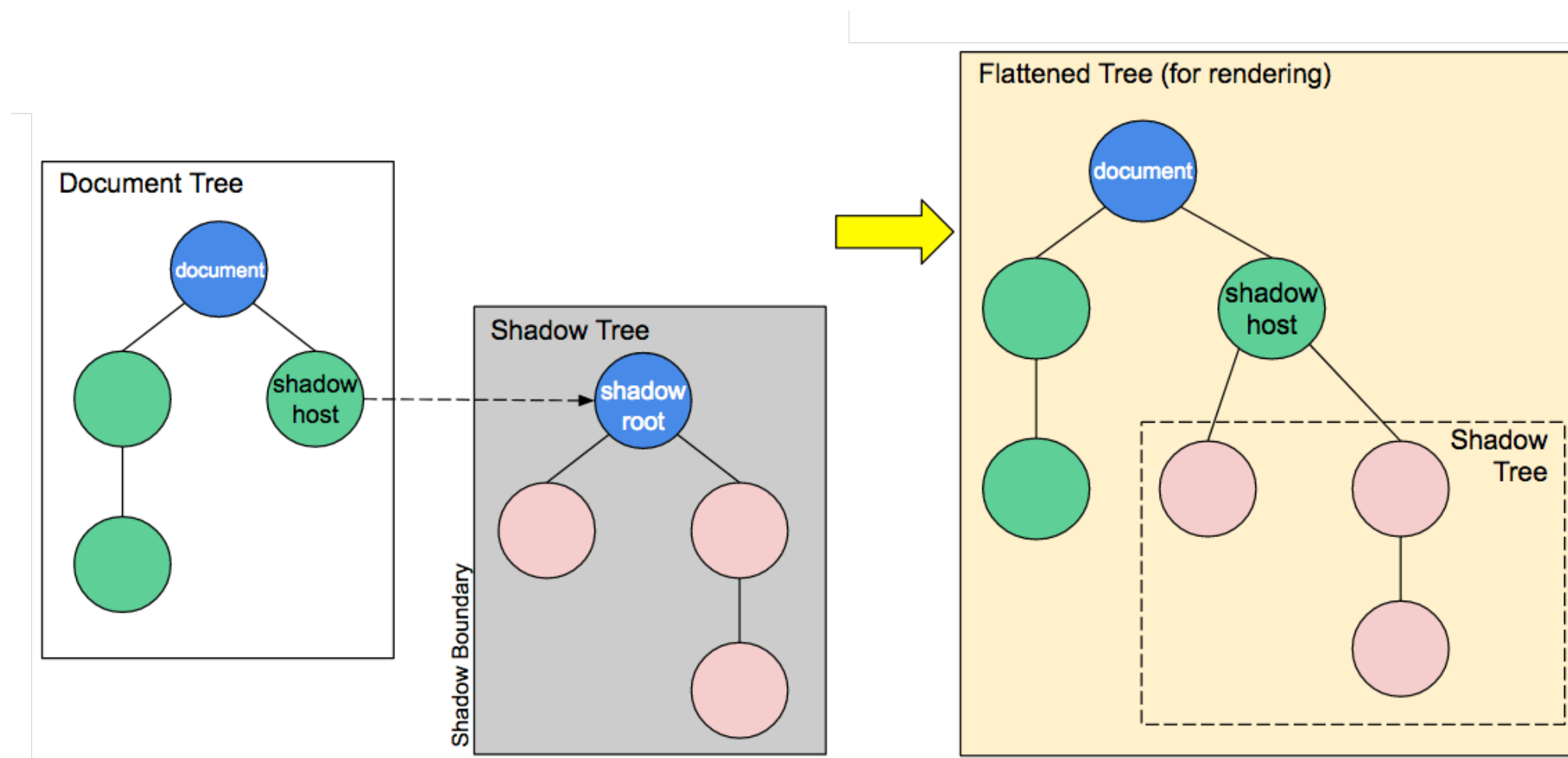
class CurrentTimeElement extends HTMLElement {
  constructor() {
    // 클래스 초기화 (필수 호출)
    super();
  }
  // 커스텀 엘리먼트가 처음 document의 DOM에 연결될 때 호출
  // 커스텀 엘리먼트 태그를 선언하면 호출
  connectedCallback() {
    // DOM에 추가되었다.
    // ...
  }
  // 사용자 정의 요소가 문서의 DOM과 연결되어 있지 않을 때 호출
  disconnectedCallback() {
    // DOM에서 제거되었다.
    // ...
  }
  // 속성의 변화를 감시 (constructor 보다 먼저 실행됨)
  // 브라우저는 observedAttributes 배열의 허용 목록에 추가된 모든 속성에 대해 attributeChangedCallback()을 호출
  static get observedAttributes() {
    // 모니터링 할 속성 이름
    return ['data-locale'];
  }
  // 속성의 변화에 반응 (속성이 추가/제거/변경)
  // 사용자 정의 요소의 속성 중 하나가 추가, 제거 또는 변경되면 호출 (observedAttributes 속성에 나열된 특성만 이 콜백을 수신)
  attributeChangedCallback(attrName, oldValue, newValue) {
    // 속성이 추가/제거/변경되었다.
    if(attrName == 'data-locale') {
      // ...
    }
  }
  // 사용자 정의 요소를 새 문서로 이동할 때 호출 (해당 엘리먼트가 다른 Document에서 옮겨져 올 때 수행)
  // 사용자설정 요소가 새 document(예: document.adoptNode(el)라고도 함)로 이동된 경우
  adoptedCallback(oldDoc, newDoc) {
    // 다른 Document 에서 옮겨져 왔다. (자주 쓸 일은 없을 것.)
    // ...
  }
}
if(!customElements.get('current-time')) {
  customElements.define("current-time", CurrentTimeElement);
}

```

# Shadow DOM

## Web Component

컴포넌트의 DOM, CSS, HTML 등을 감추는 캡슐화(encapsulation)와 외부로부터의 간섭을 제어하는 스코프(Scope)의 분리를 제공  
“DOM 안의 DOM”으로 생각할 수도 있지만, 글로벌 DOM 트리에서 완전히 분리된 고유의 요소와 스타일을 가진 DOM 트리



- shadow host : shadow DOM 이 연결된 일반 DOM node
- shadow root : shadow tree 의 root node (shadow host 지정할 때 하위로 shadow root 생성/연결)
- shadow tree : shadow DOM 의 내부 DOM tree

[Shadow DOM slots] <https://javascript.info/slots-composition>

[출처] <https://developers.google.com/web/fundamentals/web-components/shadowdom?hl=ko>

# Shadow DOM

## Web Component

HTML, CSS 캡슐화

```
<!-- shadow dom 생성/연결할 shadow host 일반 요소(node) -->
```

```
<header id="shadow-host"></header>
```

```
<!-- shadowdom 렌더링 -->
```

```
<header id="shadow-host">
```

```
  #shadow-host (closed)
```

```
  <style>
```

```
    * {
```

```
      color: #eee;
```

```
    }
```

```
  </style>
```

```
  <h1>Hello 유성민</h1>
```

```
</header>
```

```
// shadow host
```

```
let shadowHost = document.querySelector('#shadow-host');
```

```
// shadow root - mode: open or closed
```

```
let shadowRoot = shadowHost.attachShadow({mode: 'closed'});
```

```
// shadow tree
```

```
shadowRoot.innerHTML = `
```

```
  <style>
```

```
    * {
```

```
      color: #eee;
```

```
    }
```

```
  </style>
```

```
  <h1>Hello 유성민</h1>
```

```
`;
```

```
// shadow root 접근
```

```
// attachShadow({mode: 'closed'}) 경우 접근 불가, null 반환
```

```
console.log(document.querySelector('#shadow-host').shadowRoot);
```

[Shadow DOM slots] <https://javascript.info/slots-composition>

[출처] <https://developers.google.com/web/fundamentals/web-components/shadowdom?hl=ko>

# ES6 Module

## Web Component

JavaScript 캡슐화, 재사용

```
// 변수, 함수, 클래스 등을 export 할 수 있다.  
// 명명된 내보내기 (named export)  
// 동일한 이름으로 가져와야 한다.  
export { name1, name2, ..., nameN };  
export { variable1 as name1, variable2 as name2, ..., nameN };  
export function FunctionName(){...} // 함수명  
export class ClassName {...} // 클래스명  
  
// 기본 내보내기(default export)는 하나만 존재해야 한다.  
// 기본 내보내기는 어떤 이름으로 가져올 수 있다.  
export default expression;  
export { name1 as default, ... };  
export default function (...) { ... }  
export default class {...}
```

```
// module import (url, /, ./, ../)  
import name from "http://module-name.mjs";  
import * as name from "/module-name.mjs";  
import { member } from "./module-name.mjs";  
import { member as alias, member2 } from "../module-name.mjs";  
import defaultMember, { member } from "/module-name.mjs";  
import defaultMember, * as alias from "/module-name.mjs";  
import defaultMember from "/module-name.mjs";  
import "/module-name.mjs";  
  
<!-- 브라우저(html)로 불러오기 (엔트리) //-->  
<script type="module" src="./module/c.js"></script>  
<script nomodule src="./module/c.js"></script>
```

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

[ECMAScript 6, 참고] <https://hacks.mozilla.org/2015/08/es6-in-depth-modules/>

# 웹컴포넌트

## Web Component

Custom Element, Shadow DOM, Template Element 등을 활용한 컴포넌트

```
<!-- template element -->
<template id="my-paragraph">
  <link rel="stylesheet" href="reset.css" />
  <style>
    p {
      color: white;
      background-color: #666;
      padding: 5px;
    }
  </style>
  <p>My paragraph</p>
</template>
```

```
// custom element define
customElements.define('my-paragraph',
  class extends HTMLElement {
    constructor() {
      super();

      // template
      let template = document.getElementById('my-paragraph');
      let templateContent = template.content;

      // shadow dom
      let shadowRoot = this.attachShadow({mode: 'open'})
        .appendChild(templateContent.cloneNode(true));
    }
  }
)
```

# 웹컴포넌트

---

Web Component

Alex Russell,

Fronteers Conference 2011에서 처음 발표

“웹 페이지 및 웹 응용 프로그램에서 사용할 새로운 사용자 정의, 캡슐화 된 HTML 태그를 만들 수 있는 웹 플랫폼 API 세트”

(2011년, 개념)

---

구글은 2013년 웹 컴포넌트 기반(스펙)의 폴리머(Polymer) 프레임워크를 출시 (2013년, 구현체)

웹 컴포넌트는 구글의 작품, 처음 릴리즈되었을 때 다른 브라우저 벤더들과의 협의 부족, Living Standard
































[출처] <http://hacks.mozilla.or.kr/2015/08/the-state-of-web-components-web-component/>

[참고] <https://www.webcomponents.org/introduction>

# 웹컴포넌트

## Web Component

웹 컴포넌트의 도입과 사용은 더디지만 점점 확대되고 있다.

Browser support	 CHROME	 OPERA	 SAFARI	 FIREFOX	 EDGE
 HTML TEMPLATES	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 CUSTOM ELEMENTS	 STABLE	 STABLE	 STABLE	 STABLE	 POLYFILL  DEVELOPING
 SHADOW DOM	 STABLE	 STABLE	 STABLE	 STABLE	 POLYFILL  DEVELOPING
 ES MODULES	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE



# 웹컴포넌트

## Web Component

캡슐화를 위한 그밖의 API (실제 실무 적용은 어려움)

```
// iframe srcdoc 속성
// <iframe> 요소에 보일 웹 페이지의 HTML 코드를 명시
// srcdoc속성을 지원하지 않는 브라우저는 src 속성에 지정된 파일이 표시
// 마이크로소프트 브라우저는 지원하지 않는다.
var iframe = document.createElement("iframe");
iframe.srcdoc = `<p>Hello World!</p>`;
document.body.appendChild(iframe);

<!-- iframe tag 에서 사용 //-->
<iframe srcdoc="<p>Hello World!</p>" src=""></frame>
```

```
<!-- style scoped 속성 //-->
<!--
    scoped 속성이 존재하면 부모 요소에만 스타일이 적용되고,
    없다면 문서 전체에 스타일이 적용된다.
    폴리필도 없으며, 현재 대부분 브라우저에서 지원을 중단했다.
-->
<section>
  <style scoped>
    p { color: red; }
  </style>
  <p>This should be red.</p>
</section>
```

[참고] <https://developer.mozilla.org/en-US/docs/Web/API/HTMLIFrameElement/srcdoc>

[참고] <https://developer.mozilla.org/ko/docs/Web/HTML/Element/style>

# 웹컴포넌트

## Web Component

**왜 웹 표준을 알아야 하나요?** (표준 API 정보/진행상황 또는 인지 필요성)

우리는 React, Vue, Angular, Backbone.js 등 라이브러리/프레임워크 도구 잘 사용하고 있는데...

### 상호운용성 (Interoperability)

프레임워크(또는 라이브러리) 컴포넌트는 훌륭하지만 그 생태계 안에서만 훌륭할 뿐이다.

Angular 컴포넌트 안에서 React 를 (쉽게) 사용할 수 없고 반대의 경우도 그렇다.

프레임워크(또는 라이브러리)를 넘어서서 다른 기술 스택의 프로젝트에서도 동작할 것이다.

### 수명 (Lifespan)

컴포넌트가 상호운용 가능하기 때문에 더 긴 수명을 갖게 되고, 새 기술에 맞춰 재작성해야할 필요가 줄어든다.

### 가용성 (Portability)

컴포넌트가 특정 라이브러리나 프레임워크에 의존하지 않는다면, 어디에도 동작하기 때문에 도입에 대한 장벽이 상당히 낮아진다.

(사용중 또는 사용예정인 프레임워크/라이브러리와 결합되어 사용 가능)

웹 표준 API 집합을 통한 웹컴포넌트는 근본적인 문제들을 해결하는데 집중하고 있습니다.

라이브러리/프레임워크 처럼 생산성이나 어플리케이션 구조 등에 주안점이 있는 것이 아닙니다. (경쟁, 대립관계가 아님)

때문에 웹 표준 방식의 컴포넌트는 다른 프레임워크들과 상호 보완 구조에 가깝고 대체하는 관계가 아닌 것입니다.

# 웹컴포넌트

## Web Component

### 도구를 사용하기 위한 목적으로 도구를 사용하고 있지 않는가 고찰

“라이브러리/프레임워크들은 다양한 문제를 해결할 강력한 도구입니다.”

그러나 무거운 덩치는 앱을 무겁게 만들고, 리소스를 사용자에게 전가 시킬 수 있으며, 코드/빌드 종속적인 부분을 생산합니다.

“도구는 필요에 의해서 사용되어야 합니다.” (어떤 문제를 해결하고자 함인지 그 목적에 의한, 우리 환경에 따른 선택)  
트렌드, 마케팅 등 외부시각에 따른 무작정 선택보다는 우리의 개발과정에서 인지되고 있는 어려움이나 문제들은 무엇이며,  
어떤 도구를 활용하면 이를 최소화하거나 빠르게 안정화 할 수 있는지 좀 더 명확한 접근에서 선택되어야 합니다.

“새로운 방법들이 제안될수록 아이러니하게도 퍼블리셔와의 협업은 더욱더 어려워지고 있습니다.

최종 결과가 될 HTML과의 형태적 괴리감은 커지고 결합도를 예측하기 힘들 정도로 뷰는 작은 조각으로 파편화될 수 있습니다. “

WoowahanJS 만들게 된 이유 중 한 부분

고도화, 안정화, 학습비용, 협업, 성능, 고객 등 우리는 무엇을 해결하기 위한 목적으로 도구를 사용 하는가

[WoowahanJS] <http://woowabros.github.io/tools/2016/09/07/woowahan-js.html>

[Google I/O 2016에서 등장한 이 모토(Keep calm and #UseThePlatform)] <https://www.youtube.com/watch?v=J4i0xJnQUzU&feature=youtu.be>

—  
감사합니다.