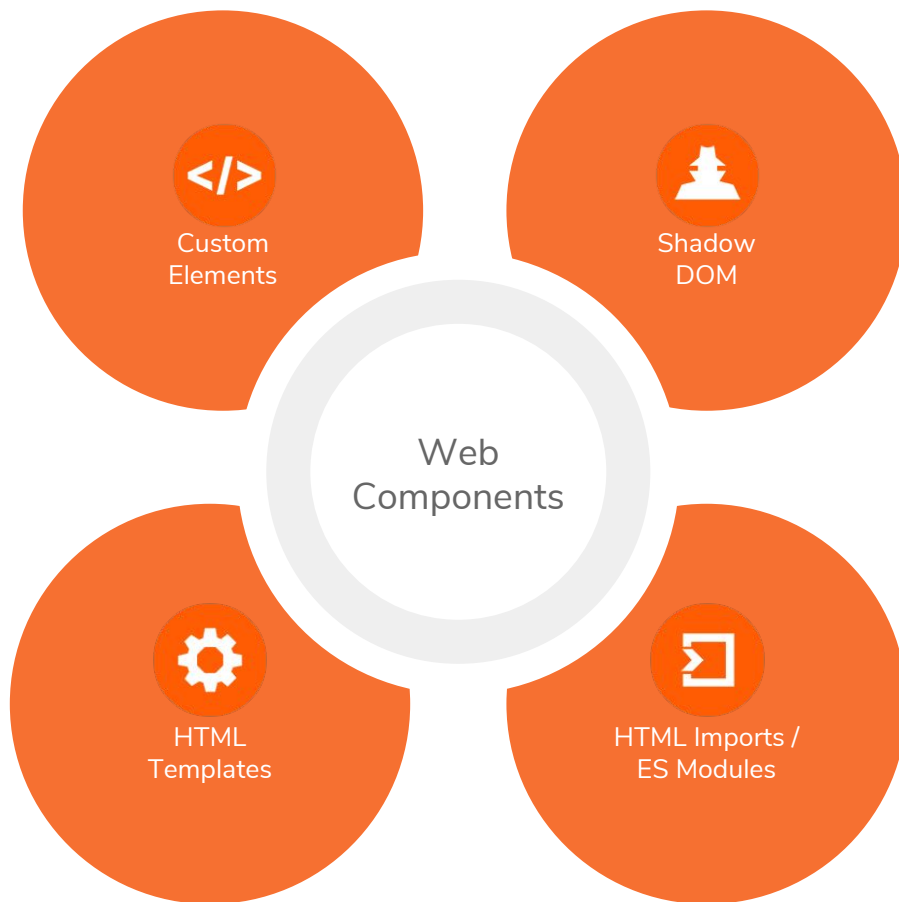




# WEB COMPONENTS













# ¿Que son los web components?



Basados en los estándares de la especificación de la W3C

# Estado de la especificación

Browser support	 CHROME	 OPERA	 SAFARI	 FIREFOX	 EDGE
 TEMPLATES	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE
 CUSTOM ELEMENTS	✓ STABLE	✓ STABLE	✓ STABLE	✓ POLYFILL • DEVELOPING	✓ POLYFILL • CONSIDERING
 SHADOW DOM	✓ STABLE	✓ STABLE	✓ STABLE	✓ POLYFILL • DEVELOPING	✓ POLYFILL • CONSIDERING
 <code>&lt;SCRIPT TYPE="MODULE"&gt;</code>	✓ STABLE	✓ STABLE	✓ STABLE	• FLAG IN 54	✓ STABLE
 HTML IMPORTS	✓ STABLE	✓ STABLE	✓ POLYFILL • ON HOLD	✓ POLYFILL • ON HOLD	✓ POLYFILL • CONSIDERING

# ¿Por qué usar Web Components?

*Que sea una solución basada en estándares web no es el único motivo por el que debemos utilizar esta tecnología en nuestros desarrollos web e híbridos.*

## Interoperabilidad



Al ser neutros los web components podemos reutilizarlos en cualquier proyecto al igual que usamos los elementos HTML.

## Sostenibilidad



Al ser módulos independientes, con un estilo y comportamiento determinado, podemos testarlos fácilmente facilitando su mantenimiento a cualquier desarrollador.

## Escalabilidad



Ampliar nuestra aplicación y añadirle nuevas funcionalidades puede ser tan sencillo como añadir nuevos componentes.

## Eficiencia en el desarrollo



Al estar libres de dependencias facilitan la reusabilidad de código incluso entre diferentes proyectos.

## Rendimiento



Son la mejor manera de construir aplicaciones web progresivas (PWA), en las que podemos cargar no solo el contenido sino también las funcionalidades de nuestra aplicación a demanda del usuario

## Simplicidad



Incluso alguien que no sepa programación debería poder construir una aplicación de manera declarativa solo con el hecho de combinar estos componentes.

# 1.

## CUSTOM ELEMENTS

*Componentes web  
reutilizables*



## Los Custom Elements nos permiten...

- Definir nuevas etiquetas HTML
- Agrupar funcionalidades propias en una única etiqueta
- Extender otros elementos
- Extender el api de otros elementos del DOM

## Registrar el nuevo elemento

- Sólo puede registrarse una vez
- El nombre debe contener un “-”

## Definir la nueva etiqueta

- No se pueden cerrar automáticamente
- Los custom elements pueden usarse antes de declararse

## Aplicar estilos

- Estilos definidos por el usuario
- Se puede aplicar un estilo preliminar al componente con la pseudoclase `:defined`

```
class AppDrawer extends HTMLElement {...}  
window.customElements.define('app-drawer', AppDrawer);  
  
// Or use an anonymous class if you don't want a named constructor in  
currentscope.  
window.customElements.define('app-drawer', class extends HTMLElement {...});
```

```
<app-drawer></app-drawer>
```

```
<!-- user-defined styling -->  
<style>  
  app-drawer {  
    display: flex;  
  }  
  app-drawer:not(:defined) {  
    /* Pre-style, give layout, replicate app-drawer's eventual styles, etc. */  
  }  
</style>
```

“

*El uso de un elemento personalizado no difiere del uso de un <div> u otro elemento. Las instancias pueden declararse en la página, **crearse de forma dinámica** en JavaScript y **tomar receptores de eventos** como adjuntos, entre otras posibilidades*



# Definición del API JavaScript de un elemento

- Se define mediante una **clase de ES6** que extiende de **HTMLElement**
- La extensión de **HTMLElement** permite que se herede todo el API del DOM
- Se definen los nuevos atributos mediante **get y set**
- Se definen los nuevos **métodos**
- El **this** dentro del elemento hacer referencia al componente

```
class AppDrawer extends HTMLElement {  
  // A getter/setter for an open property.  
  get open() {  
    return this.hasAttribute('open');  
  }  
  set open(val) {  
    // Reflect the value of the open property as an HTML attribute.  
    (val) ? this.setAttribute('open', '') : this.removeAttribute('open');  
    this.toggleDrawer();  
  }  
  // A getter/setter for a disabled property.  
  get disabled() {  
    return this.hasAttribute('disabled');  
  }  
  set disabled(val) {  
    // Reflect the value of the disabled property as an HTML attribute.  
    (val) ? this.setAttribute('disabled', '') : this.removeAttribute('disabled');  
  }  
  // Can define constructor arguments if you wish.  
  constructor() {  
    // If you define a ctor, always call super() first!  
    // This is specific to CE and required by the spec.  
    super();  
    // Setup a click listener on <app-drawer> itself.  
    this.addEventListener('click', e => {  
      // Don't toggle the drawer if it's disabled.  
      if (this.disabled) { return; }  
      this.toggleDrawer();  
    });  
  }  
  toggleDrawer() {  
    ...  
  }  
}  
customElements.define('app-drawer', AppDrawer);
```

## Extender un elemento personalizado

```
class FancyDrawer extends AppDrawer {  
  
  constructor() {  
    super(); // Always call super() first in the ctor.  
             // This also calls the extended class' ctor.  
    ...  
  }  
  toggleDrawer() {  
    // Possibly different toggle implementation?  
    // Use ES2015 if you need to call the parent method.  
    // super.toggleDrawer()  
  }  
  anotherMethod() { ... }  
}  
customElements.define('fancy-app-drawer', FancyDrawer);
```

## Extender elemento HTML nativo

- En lugar de extender de `HTMLElement` extenderá de la extensión correcta del DOM (`HTMLButtonElement` ....)

- El `define` lleva un tercer parámetro con el elemento al que extiende

- Se usa con la etiqueta nativa con el atributo `is`

```
class FancyButton extends HTMLButtonElement {  
  constructor() {  
    super(); // always call super() first in the ctor.  
    this.addEventListener('click', e => this.drawRipple(e.offsetX, e.offsetY));  
  }  
  // Material design ripple animation.  
  drawRipple(x, y) { ... }  
}  
customElements.define('fancy-button', FancyButton, {extends: 'button'});  
  
<button is="fancy-button" disabled>atractivo botón.</button>
```

# Ciclo de vida de los componentes

Nombre	Momento en el que se invoca
<b>constructor</b>	Se crea o <a href="#">se actualiza</a> una instancia del elemento. Es útil para inicializar el estado, configurar receptores de eventos o <a href="#">crear un Shadow DOM</a> . Consulta la especificación para obtener información sobre las restricciones en relación con lo que puedes hacer en el constructor.
<b>connectedCallback</b>	Se llama cada vez que se inserta el elemento en el DOM. Es útil para ejecutar código de configuración, como la obtención de recursos o la representación. En general, debes intentar demorar el trabajo hasta este momento.
<b>disconnectedCallback</b>	Se llama cada vez que se quita el elemento del DOM. Es útil para ejecutar código de limpieza (eliminación de receptores de eventos, etc.).
<b>attributeChangedCallback(attrName, oldVal, newVal)</b>	Se agrega, quita, actualiza o reemplaza un atributo. También se llama para obtener valores iniciales cuando el analizador crea un elemento o lo <a href="#">actualiza</a> . Nota: solo los atributos que se indiquen en la propiedad observedAttributes recibirán este callback.
<b>adoptedCallback()</b>	El elemento personalizado se traslada a un nuevo document (p. ej., alguien llama a document.adoptNode(el)).

# Propiedades y atributos

## Reflejar propiedades en atributos

- Los atributos también son útiles para configurar un elemento de forma declarativa.
- Ciertas API, como los selectores de CSS y accesibilidad, depende de los atributos.
- Conservar la representación del elemento DOM en sincronización con su estado de JavaScript.

```
...  
  
get disabled() {  
  return this.hasAttribute('disabled');  
}  
  
set disabled(val) {  
  // Reflect the value of `disabled` as an attribute.  
  if (val) {  
    this.setAttribute('disabled', '');  
  } else {  
    this.removeAttribute('disabled');  
  }  
  this.toggleDrawer();  
}
```

# Observar cambios en los atributos

- Los componentes pueden reaccionar a los cambios que se produzcan en los atributos a través del método **attributeChangeCallback**.

- El navegador llamará a este método para cada atributo que esté definido en el array **observedAttributes**.

```
class AppDrawer extends HTMLElement {  
  ...  
  
  static get observedAttributes() {  
    return ['disabled', 'open'];  
  }  
  
  get disabled() {  
    return this.hasAttribute('disabled');  
  }  
  
  set disabled(val) {  
    if (val) {  
      this.setAttribute('disabled', '');  
    } else {  
      this.removeAttribute('disabled');  
    }  
  }  
  
  // Only called for the disabled and open attributes due to observedAttributes  
  attributeChangedCallback(name, oldValue, newValue) {  
    // When the drawer is disabled, update keyboard/screen reader behavior.  
    if (this.disabled) {  
      this.setAttribute('tabindex', '-1');  
      this.setAttribute('aria-disabled', 'true');  
    } else {  
      this.setAttribute('tabindex', '0');  
      this.setAttribute('aria-disabled', 'false');  
    }  
    // TODO: also react to the open attribute changing.  
  }  
}
```

# 2.

## SHADOW DOM

*Componentes web  
independientes*



# ¿Que es el shadow DOM?

- **DOM aislado:** El DOM de un componente es autocontenido (p.ej., `document.querySelector()` no muestra nodos en el shadow DOM del componente).
- **CSS con ámbito:** la CSS definida dentro de shadow DOM está acotado al DOM. Las reglas de estilo no filtran y los estilos de página no se infiltran.
- **Composición:** Diseña una API declarativa basada en lenguaje de marcado para tu componente.
- **Simplifica CSS:** el DOM dentro del ámbito significa que puedes usar simples selectores de CSS, nombres de id/clase más genéricos, y no preocuparte por conflictos de nombres.
- **Productividad:** piensa en apps en fragmentos del DOM en lugar de una gran página (global).

## Como crear un shadow DOM

- Algunos elementos no pueden alojar un shadow tree, o bien porque ya lo tienen (<textarea>, <input>) o bien porque no tiene sentido (<img>)

## Shadow DOM para un custom element

```
const header = document.querySelector('header');
const shadowRoot = header.attachShadow({mode: 'open'});
shadowRoot.innerHTML = '<h1>Hello Shadow DOM</h1>'; // Could also use appendChild().
// header.shadowRoot === shadowRoot
// shadowRoot.host === header
```

```
// Use custom elements API v1 to register a new HTML tag and define its JS behavior
// using an ES6 class. Every instance of <fancy-tab> will have this same prototype.
customElements.define('fancy-tabs', class extends HTMLElement {
  constructor() {
    super(); // always call super() first in the ctor.
    // Attach a shadow root to <fancy-tabs>.
    const shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML = `
      <style>#tabs { ... }</style> <!-- styles are scoped to fancy-tabs! ->
      <div id="tabs">...</div>
      <div id="panels">...</div>    `;
  }
  ...
});
```



# Shadow DOM

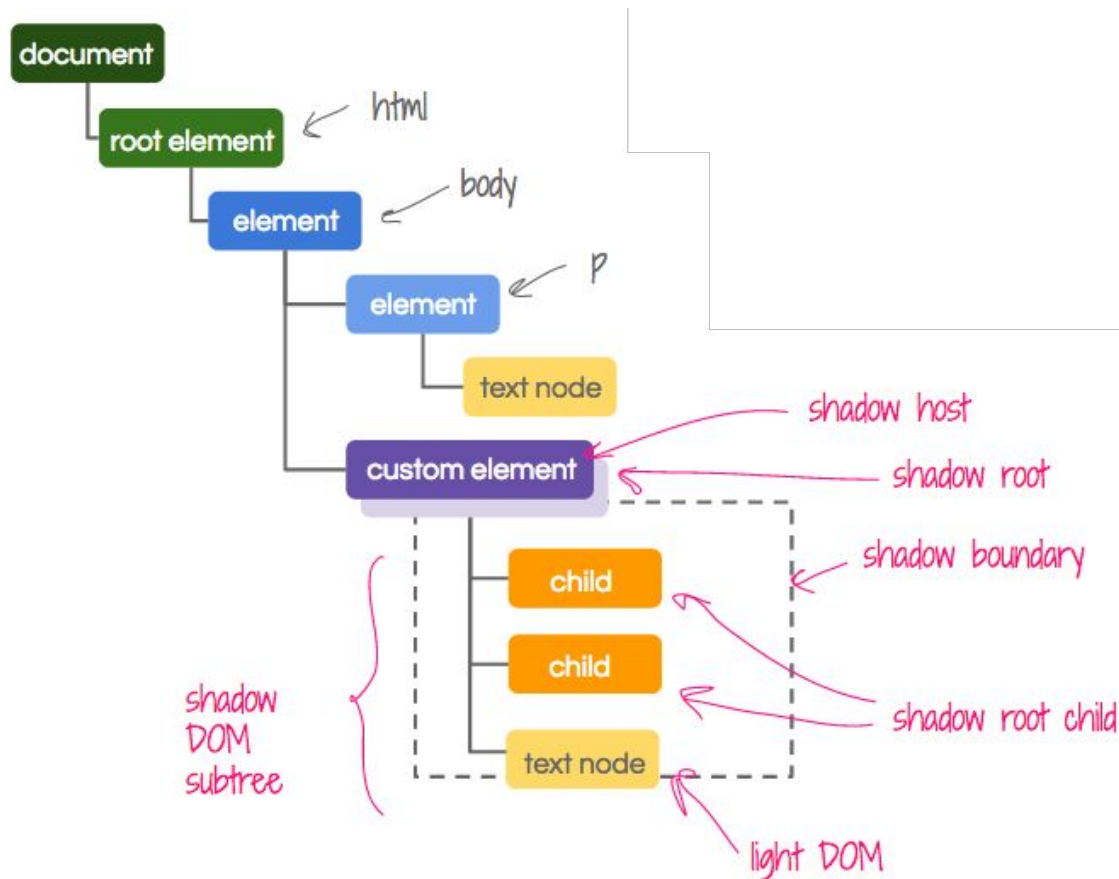
- Es local del componente y define su estructura interna, CSS con ámbito, y encapsula tus detalles de implementación.

## Light DOM

- Es el lenguaje de marcado que escribe un usuario de tu componente. El DOM vive afuera del shadow DOM del componente.

## Arbol DOM plano

- El resultado de que el navegador distribuya el light DOM del usuario en tu shadow DOM, representando el producto final. El árbol plano es lo que en última instancia ves en las DevTools y lo que se representa en la página.



## Elemento <SLOT>

- Los slots son una forma de crear una "API declarativa" para un componente web. Se mezclan en el DOM del usuario para ayudar a representar el componente general, **componiendo así distintos DOM trees juntos**.

## <SLOT> con nombre

- Podemos definir todos los slot que queramos a nuestro componente, asignándole un name diferente
- El árbol aplanado resultante será el mismo independientemente del orden en que hayamos generado el light dom

```
<!-- Default slot. If there's more than one default slot, the first is used.-->

<slot></slot>
<slot>Fancy button</slot> <!-- default slot with fallback content -->
<slot> <!-- default slot entire DOM tree as fallback -->
  <h2>Title</h2>
  <summary>Description text</summary>
</slot>
```

```
#shadow-root
<div id="tabs">
  <slot id="tabsSlot" name="title"></slot>
</div>
<div id="panels">
  <slot id="panelsSlot"></slot>
</div>
<fancy-tabs>
  <button slot="title" selected>Title</button> //<h2 slot="title">Title</h2>
  <button slot="title">Title 2</button> //<section>content panel 1</section>
  <button slot="title">Title 3</button> //<h2 slot="title">Title</h2>
  <section>content panel 1</section> //<section>content panel 1</section>
  <section>content panel 2</section> //<h2 slot="title">Title</h2>
  <section>content panel 3</section> //<section>content panel 1</section>
</fancy-tabs>
```

# Estilos

- Los selectores de CSS que se usan dentro de un shadow DOM influyen en tu componente de forma local.
- Las reglas de la página principal tienen más especificidad que las reglas de **:host**
- `:host(<selector>)` te permite apuntar al host si coincide con un `<selector>`
- Usa la contención de CSS ( `contain: content` ) para mejorar el resultado
- Usa `all: initial;` para restablecer los estilos heredable a su valor inicial cuando cruzan el límite de shadow, ya que los estilos heredables (`background`, `color`, `font`, `line-height`, etc.) continúan heredando en shadow DOM

```
<style>
:host {
  all: initial; /* 1st rule so subsequent properties are reset. */
  opacity: 0.4;
  will-change: opacity;
  transition: opacity 300ms ease-in-out;
  contain: content; /* Boom. CSS containment FTW. */
}
:host(:hover) {
  opacity: 1;
}
:host([disabled]) { /* style when host has disabled attribute. */
  background: grey;
  pointer-events: none;
  opacity: 0.4;
}
:host(.blue) {
  color: blue; /* color host when it has class="blue" */
}
:host(.pink) > #tabs {
  color: pink; /* color internal #tabs node when host has class="pink". */
}
#tabs {
  box-shadow: 0 2px 2px rgba(0, 0, 0, .3);
  background: white;
  ...
}
#panels {
  display: inline-flex;
}
</style>

<div id="tabs"> ... </div>
<div id="panels"> ... </div>
```

## Estilos basados en contexto

- Útil para aplicar temas

```
<style>
  :host-context(.darktheme) {
    color: white;
    background: black;
  }
</style>
<body class="darktheme">
  <fancy-tabs> ... </fancy-tabs>
</body>
```

## Estilos de nodos distribuidos

- Solo se puede aplicar estilos a los hijos directos

```
<name-badge>
  <h2>Eric Bidelman</h2>
  <span class="title">
    Digital Jedi, <span class="company">Google</span>
  </span>
</name-badge>

<style>
  ::slotted(h2) {
    margin: 0;
    font-weight: 300;
    color: red;
  }
  ::slotted(.title) {
    color: orange;
  }
</style>
<slot></slot>
```

*/\* DOESN'T WORK (can only select top-level nodes).  
::slotted(.company),  
::slotted(.title .company) {  
 text-transform: uppercase;  
}\*/*

# 3.

HTML  
TEMPLATES





*El elemento <template> permite declarar fragmentos de DOM que se analizan, permanecen inactivos durante la carga de la página y se pueden activar más adelante en el tiempo de ejecución. Es otra primitiva de API de la familia de componentes web. **Las plantillas son marcadores de posición ideales para declarar la estructura de un elemento personalizado.***

## El contenido dentro de la etiqueta `<template>`

- No se renderiza hasta que es activado
- No tiene efectos en otra parte de la página.
  - Los `<script>` no se ejecutan
  - Las imágenes no cargan
  - Los `<audio>/<video>` no se inician
- No aparece en el DOM

Registrar un elemento con contenido de Shadow DOM creado a partir de un elemento `<template>`

```
<template id="x-foo-from-template">
  <style>
    p { color: orange; }
  </style>
  <p>I'm in Shadow DOM. My markup was stamped from a &lt;template&gt;.</p>
</template>
<script>
  customElements.define('x-foo-from-template', class extends HTMLElement {
    constructor() {
      super(); // always call super() first in the ctor.
      let shadowRoot = this.attachShadow({mode: 'open'});
      const t = document.querySelector('#x-foo-from-template');
      const instance = t.content.cloneNode(true);
      shadowRoot.appendChild(instance);
    }
    ...
  });
</script>
```



# 4.

HTML  
IMPORT / ES  
MODULES



# HTML Import

*Define la inclusión y reutilización de un documento HTML en otro documento HTML*

```
<link rel="import" href="cool-thing.html">  
<cool-thing> </cool-thing>
```

# ES MODULES

Exportar el  
componente

```
export default class AppDrawer extends HTMLElement { ... }
```

Importar en el  
archivo donde se  
vaya a usar

```
<script type="module">  
  import AppDrawer from "../app-drawer.js";  
</script>
```

- Definir un script de **tipo “module”**
- Importar componente

# 5.

Buenas  
prácticas



# Shadow DOM

## Crear Shadow root para encapsular los estilos

Encapsular los estilos del elemento asegura que se verá bien independientemente de donde se utilice.

## Crear Shadow root en el constructor

Encapsular los estilos del elemento asegura que se verá bien independientemente de donde se utilice.

## Asignar un display al :host del elemento

Los custom elements tienen *display:inline* por defecto. Es importante ponerle otro display para evitar problemas de maquetación a otros desarrolladores.

## Definir un estilo en el :host para el atributo hidden

# Atributos y propiedades

## No sobrescribir los atributos globales del autor del componente

Los atributos globales son los que están presentes en todos los elementos HTML(role, tab-index....). Si por ejemplo quisiéramos poner a un elemento un tab-index de 0, es conveniente mirar si el autor no lo setea a -1 ya que con esto estaría diciendo que no quiere que sea “focusable”.

## Asociar propiedades y atributos

Lo ideal es que cada atributo este linkado a su propiedad correspondiente

## Mantener sincronizados los atributos con las propiedades

## Recibir datos primitivos tanto por atributos como por propiedades

## Recibir datos complejos solo por propiedades

## No reflejar datos complejos en atributos

## No aplicar clases

Las clases generalmente deben de ser usadas por el desarrollador que utiliza el componente, por lo que, al usarlas nosotros podemos pisar esas clases

# Eventos

---

## Disparar eventos como respuesta a actividad interna del componente

Si dentro del componente hay alguna actividad que solo conoce este (animación, temporizador, carga asíncrona...) y que produce algún cambio en una propiedad, es conveniente notificar este cambio al exterior mediante un evento.

## Declarar los listeners en el `connectedCallback` y eliminarlos en el `disconnectedCallback`

# Referencias

- [Custom Elements V1](#) **Google Developers**
- [Shadow DOM](#) **Google Developers**
- [Web components](#) **webcomponents.org**
- [Como crear un web component de forma nativa](#) **Carlos Azaustre**