POLYMER 2.*

“

#UseThePlatform

Novedades en Polymer 2.0

- Su peso en bytes es aproximadamente un cuarto del peso de Polymer 1.0
- Los polyfills están separados por especificaciones del estándar de Web Components.
- Los Custom Elements ahora se definen mediante **clases ES6**. En el caso de crear un elemento de Polymer 2.0 la única diferencia está en que no extenderemos de HTMLElement sino de Polymer.Element.
- La interoperabilidad entre componentes, la capacidad de trabajar los unos con los otros, ahora está directamente ofrecida con el estándar de Web Components.
- El uso de Custom Properties de CSS se implementa de manera nativa y compatible con todos los navegadores modernos, lo que no requiere código extra de la librería.
- Existe una capa de compatibilidad que permite el desarrollo híbrido, de componentes con sintaxis 1.x y con sintaxis 2.x, de modo que la migración de aplicaciones existentes sea más sencilla y se pueda realizar de manera progresiva.

Polymer 1.9 + Polyfill size	40k	53k	53k	53k	53k
Polymer 2.0 + Polyfill size	11k	14k	33k	33k	37k

ESTRUCTURA DE COMPONENTES

Estructura de un componente

Importar polymer-element

```
<link rel="import"  href="/bower-components/polymer/polymer-element.html">
```

Etiqueta dom-module

```
<dom-module id="nombre-componente"></dom-module>
```

- Componente base para crear componentes de polymer 2
- Va a englobar todo nuestro componente.
- Debe llevar un id con el nombre del componente.

Template del componente

```
<template>
 <h2>Esto es el HTML del componente</h2>
</template>
```

- Dentro colocaremos el código HTML de nuestro componente
- Las etiquetas que coloquemos en el template formarán parte del shadow dom del componente

Estructura de un componente

Estilos del componente

```
<template>
 <style>
 h2 {
   background-color: orange;
   color: white;
 }
 </style>
 <h2>Esto es el HTML del componente</h2>
</template>
```

- Los componentes de Polymer 2 pueden tener sus propios estilos
- El CSS va dentro de la etiqueta template
- Los estilos de nuestro componente no se verán afectados por los estilos ya definidos para el H2 . Tampoco los estilos del componente entrarán en conflicto con los del resto de la página donde se aloje.

Estructura de un componente

Declaración del componente

```
class NombreComponente extends Polymer.Element {
   static get is() {
     return 'nombre-componente';
   }
   constructor() {
     super();
   }
}
```

- Definimos la clase que implementará nuestro componente
- El componente irá nombrado con un getter en la propiedad “is”

Definición del componente

```
customElements.define(NombreComponente.is, NombreComponente);
```

Estructura de un componente

Componente final

```
<link rel="import"  href="bower-components/polymer/polymer-element.html">
<dom-module id="nombre-componente">
 <template>
   <style>
   h2 {
     background-color: orange;
     color: white;
   }
   </style>
   <h2>Esto es el HTML del component</h2>
 </template>
 <script>
 class NombreComponente extends Polymer.Element {
   static get is() { return 'nombre-componente';}
 }
 customElements.define(NombreComponente.is, NombreComponente);
 </script>
</dom-module>
```

Uso del componente

```
<link rel="import" href="elementos/navegador-secciones.html">
<nombre-componente></nombre-componente>
```

CICLO DE VIDA

Ciclo de vida de los componentes

Nombre	Momento en el que se invoca
constructor	Es el propio constructor de la clase ES6 que se escribe para implementar un custom element. El constructor se invoca una vez por cada elemento singular creado de un tipo.
connectedCallback	Este estado del ciclo de vida ocurre cuando el elemento se inyecta en el DOM de la página.
disconnectedCallback	Este estado ocurre cuando el elemento se borra del DOM de la página.
attributeChangedCallback(attrName, oldVal, newVal)	Este momento del ciclo de vida de los componentes se produce cada vez que un atributo o propiedad del componente cambia.
ready()	Este callback para definir acciones durante el ciclo de vida se denomina "One-time initialization" en la documentación de Polymer 2. Básicamente es como un "connectedCallback", solo que no se ejecuta más que 1 vez, la primera que se inyecta en el DOM de la página.

- Los callbacks del ciclo de vida son opcionales.
- Cada callback debe invocar al método correspondiente de su clase padre.

PROPIEDADES

“

En Polymer 2, existe la posibilidad de trabajar de una manera ágil con propiedades de los componentes, así como asignar valores desde fuera de éstos, de manera declarativa en el propio HTML.

Las propiedades, que nos permitirán manejar el estado de los componentes.

Declaración de las propiedades

Para declarar propiedades, añadiremos el get de una propiedad estática a la clase del elemento. Este get debe devolver un objeto que contenga la declaración de las propiedades.

```
class EstadoAnimo extends Polymer.Element {
 static get is() {
   return 'estado-animo';
 }
 static get properties() {
   return {
     animo: {
       type: String,
       value: 'feliz'
     }
   }
 }
}
customElements.define(EstadoAnimo.is, EstadoAnimo);
```

Las propiedades se pueden reflejar en la representación del componente.

```
<template>
  <div>
    Hoy me encuentro <span>[[animo]]</span>
  </div>
</template>
```

Propiedades de las propiedades

LLave	Detalle
type	Type: constructor El tipo del atributo es usado para su deserialización. Polymer soporta la deserialización de los siguientes tipos Boolean, Date, Number, String, Array y Object.
value	Type: boolean, number, string or function. Valor por defecto de la propiedad. Si la propiedad es un objeto o un array, se debe usar una función para asegurar que cada elemento tiene su propia copia y no va a tener un objeto u array compartido entre todas las instancias.
reflectToAttribute	Type: boolean. Cuando es true provoca que la correspondiente propiedad se refleje en el nodo del componente como un atributo. Si la propiedad es un booleano se mostrará como un atributo booleano de HTML. Para los demás tipos se usará la representación en string de su valor.
readOnly	Type: boolean. Cuando es true la propiedad no puede ser asignada directamente

Propiedades de las propiedades

LLave	Detalle
notify	Type: boolean. Cuando es true provoca que la correspondiente propiedad se refleje en el nodo del componente como un atributo.
computed	Type: string. El valor será un string con la invocación del método con la lista de propiedades de las que depende como atributo. Una propiedad computada irá en función de los valores de una o más de las otras propiedades, recalculándose cada vez que una de estas cambia. Las propiedades computadas son siempre de solo lectura.
observer	Type: string. El valor es interpretado como un método que se invocará cada vez que el valor de la propiedad cambie.

Propiedades y atributos

- Atributo: Valor que se coloca en la etiqueta HTML.
- Propiedad: Valor interno del JavaScript del componente.
- Para Polymer todas las propiedades tendrán su atributo correspondiente.
- Los atributos al ser HTML no diferencian entre mayúsculas y minúsculas, por lo que se utilizan guiones para separar las palabras. Mientras que las propiedades javascript equivalentes irán en camelCase.
- Las propiedades declaradas implícitamente no pueden ser configuradas desde el HTML.

Deserialización

El proceso por el que se traspasan los valores del HTML a las propiedades es la deserialización y en él se intentará corresponder el valor del atributo con el tipo de dato indicado en la declaración de la propiedad.

Esto es importante porque todos los datos colocados como valores de atributos en el HTML son cadenas de texto. HTML no distingue si lo que se ha colocado como valor de un atributo es una cadena, un número o un booleano y sin embargo, Polymer nos hará el trabajo de corresponder y transformar aquella información al tipo correcto.

Tipos de datos en la deserialización

- **Cadenas (String)**: Aquí Polymer no realiza ningún trabajo especial, puesto que los valores de los atributos son siempre cadenas.
- **Números (Number)**: Polymer se encarga de convertir la cadena en un valor numérico, haciendo lo posible porque esa conversión sea lógica. Convierte por ejemplo la cadena "4" por el número 4. Convierte la cadena "092" por el número 92. Aunque algo como "x665" no lo podría convertir en un número y otorgará a la propiedad el valor "NaN" (not a number).
- **Boleanos (Boolean)**: En este caso no importa el valor del atributo, sino simplemente su presencia. En el caso que el atributo se encuentre en la etiqueta, entonces la propiedad tendría valor asignado como "true". Si el atributo no está presente, la propiedad nacerá con el valor "false" (a no ser que se le indique otro valor predeterminado).
- **Arrays y objetos (Array / Object)**: Para estos dos tipos de datos los valores de los atributos se tienen que indicar con notación JSON. En este caso es importante tener en cuenta que en un JSON las comillas válidas son las comillas dobles (") y por tanto no nos queda otro remedio de colocar el valor del atributo HTML con comillas simples. Luego verás un ejemplo.
- **Fechas (Date object)**: Polymer tratará de convertir la fecha en un objeto de la clase Date de Javascript. Ten en cuenta que debes usar un formato de fecha apropiado como valor de atributo, como 2017-09-01.

Deserialización personalizada

Una utilidad interesante en Polymer 2 es la deserialización personalizada, que permite al desarrollador sobreescribir el procedimiento para deserializar un valor de un tipo de atributo, ya sea de uno de los ya definido o de uno creado por nosotros mismos.

```
class TipoPersonalizado {
  //aquí el código de un nuevo tipo inventado, definido por medio de una clase ES6
}
```

```
static get properties() {
 return {
   miFechaEspanola: TipoPersonalizado
 };
}
```

```
_deserializeValue(value, type) {
 if (type == Date) {
  return this.fechaEspanolaToDate(value);
 }
 if (type = TipoPersonalizado) {
  return this.parsearOtraCosa(value);
 }
 return super._deserializeValue(value, type);
}
```

DATA-BINDING

“

Simplemente es un enlace, para asociar un mismo dato a diversos integrantes de una aplicación, de tal modo que si uno de ellos lo cambia, su valor cambie en los otros elementos que tienen ese dato bindeado.

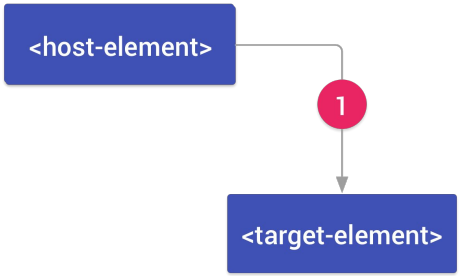
Tipos de binding

El data-binding conecta los datos de un custom element (host) a las propiedades o atributos de su DOM local. Los datos del host pueden ser una propiedad o un sub-propiedad representada por su “data path”.

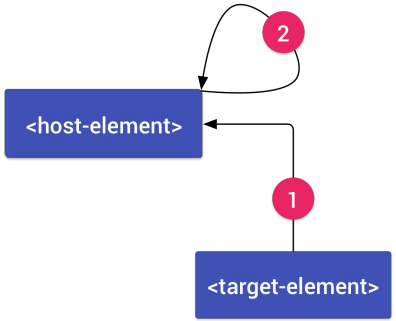
El data-binding se crea añadiendo anotaciones en el templete.

Tipos de anotaciones

ONE-WAY Binding:
El binding de una sola dirección "[[]]" . En este caso el valor bindeado viaja del padre al hijo pero no del hijo al padre.



TWO-WAY o automatic Binding:
Binding de dos direcciones "{{}}" . Es decir, el valor bindeado viaja del padre al hijo y también del hijo al padre. La propiedad debe estar definida con notify: true



Diferentes maneras de enlazar datos

El data-binding además de conectar elementos por medio de un mismo dato, tambien supone un representación en el local DOM del componente de los mismos.

Esta conexión se puede realizar de varias maneras:

- A través de una propiedad

```
<user-view first-name="{{managerName}}"></user-view>
```

- Representación directa en el template

```
<dom-module id="user-view">
   <template>
     <div>[[name]]</div>
   </template>
    <script>
     class UserView extends Polymer.Element {
       static get is() {return 'user-view'}
       static get properties() {
         return {name: String}
       }
     }
     customElements.define(UserView.is, UserView);
   </script>
 </dom-module>
  <!-- usage -->
 <user-view name="Samuel"></user-view>
```

Diferentes maneras de enlazar datos

- A través de un atributo
  - Sólo se pueden hacer de una dirección (one way binding). Para hacer de dos direcciones hay alternativas en algunos casos.
  - Internamente Polymer los implementa con el método nativo setAttribute()
  - En algunos navegadores puede que funcione el binding a atributos sin colocar el "$", pero no te confunda, pues en otros browsers puede no funcionar.

```
<a href$="{{hostProperty}}">
```

Diferentes maneras de enlazar datos

Atributos nativos que no soportan el bindeo de propiedades

Atributo	Propiedad	Nota
class	**classList, className**	Mapea dos propiedades con diferente formato.
style	**style**	Por especificación style es considerado una referencia de sólo lectura al objeto CSSStyleDeclaration
href	**href**	
for	**htmlFor**	
data-*	**dataset**	Los atributos personalizados (atributos que el nombre empiza por data-) son almacenados en la propiedad dataset.
value	**value**	Sólo para <input type="Number">

Más sbre data-binding

Binding de sub-propiedades

Las anotaciones de binding también pueden referenciar al path de una sub-propiedad.

```
<user-view first="{{user.first}}" last="{{user.last}}"></user-view>
```

Para ver reflejados los cambios en una sub-propiedad lo haremos de la siguiente manera

```
// DOES NOT WORK
this.profile.name = Alex;
// DOES WORK
this.set('profile', this.profile);
// OR
this.profile.name = Alex;
this.notifyPath('profile.name');
```

Operadores lógicos

Las anotaciones de binding soportan la inclusión del operador de negación (!) como primer carácter.

```
<my-page show-login="[[!isLoggedIn]]"></my-page>
```

Más sbre data-binding

Binding computado

Si el resultado de lo que queremos mostrar depende de operaciones lógicas más o menos complejas podemos invocar a una función dentro de nuestras anotaciones de binding

```
<div>[[_formatName(first, last, title)]]</div>
```

Al igual que los observers y las propiedades computadas, las funciones no se invocan **hasta que todos sus argumentos estén definidos**.

Composición de bindings

Se pueden combinar tantas propiedades bindeadas como necesitemos, tanto en un contenido de texto como en un string literal.

```
<img src$="https://www.example.com/profiles/[[userId]].jpg">
 <span>Name: [[lastname]], [[firstname]]</span>
```

Más sbre data-binding

Binding sobre arrays

Polymer tiene disponible una serie de de métodos para modificar los arrays.

- push(path, item1, [..., itemN])
- pop(path)
- unshift(path, item1, [..., itemN])
- shift(path)
- splice(path, index, removeCount, [item1, ..., itemN])

En caso de utilizar las operaciones nativas de array el cambio se debe notificar con notifySplices

```
// mutate the array
this.unshift('myArray', { name: 'Susan' });
// Also we can use the set method
this.set('users.3', {name: 'Churchill'});
```

Cambiar múltiples propiedades

```
this.setProperties({
 date: 'Jan 17, 2017',
 verified: true
}) //setProperties supports an optional setReadOnly flag as the second
parameter. If you need to set read-only properties as part of a batch change,
pass true for the second parameter:
```

Más sobre data-binding

Binding sobre un elemento HTML

El data binding sobre un elemento nativo que no sigue la convención de nombres, se puede especificar con un evento personalizado.

target-prop="{{hostProp::target-change-event}}"

```
<!-- Listens for `input` event and sets hostValue to <input>.value -->
<input value="{{hostValue::input}}">

<!-- Listens for `change` event and sets hostChecked to <input>.checked -->
<input type="checkbox" checked="{{hostChecked::change}}">

<!-- Listens for `timeupdate ` event and sets hostTime to <video>.currentTime -->
<video url="..." current-time="{{hostTime::timeupdate}}">
```

OBSERVERS

Observers simples

Los observers simples se declaran en el objeto de la propiedad.
Estos son disparados en cada cambio de valor de la propiedad
Para declarar un observer se debe especificar el nombre de la función a la que se invoca, la cual debe estar en el API del componente.
El método del observer recibe como argumentos el nuevo valor y el antiguo de la propiedad

```
class XCustom extends Polymer.Element {
 static get is() {return 'x-custom'; }
 static get properties() {
   return {
     active: {
       type: Boolean,
       // Observer method identified by name
       observer: '_activeChanged'
     }
   }
 }

 // Observer method defined as a class method
 _activeChanged(newValue, oldValue) {
   this.toggleClass('highlight', newValue);
 }
}
```

Observers complejos

Los observers complejos se declaran en el array de observers.
Estos son disparados en cada cambio de valor de una propiedad
Estos observers evalúan un conjunto de propiedades
Los observers múltiples no devuelven el valor anterior, sólo el nuevo

```
class XCustom extends Polymer.Element {
 static get is() {return 'x-custom'; }
 static get properties() {
   return {
       preload: Boolean,
       src: String,
       size: String
   }
 }
 // Each item of observers array is a method name followed by
 // a comma-separated list of one or more dependencies.
 static get observers() {
   return [
       'updateImage(preload, src, size)'
   ]
 }
 // Each method referenced in observers must be defined in
 // element prototype. The arguments to the method are new value
 // of each dependency, and may be undefined.
 updateImage(preload, src, size) {
   // ... do work using dependent values
 }
}
```

Observers en sub-propiedades

Igual que los observers complejos se declaran en el array de observers .
Para que se reflejen los cambios en la propiedad debemos realizarlos obligatoriamente con el método set().

```
static get observers() {
  return [
    'userNameChanged(user.name)'
  ]
}
userNameChanged: function(name) {
  console.log('new name: ' + name);
  this.set("user.name", name);
}
```

Si queremos observar todas las propiedades de un objeto lo indicamos con *.
En este caso en la función recibiremos un objeto con el path (ruta de la propiedad que cambio) y el value (nuevo valor)

```
static get observers() {
  return ['addressChanged(user.address.*)']
}

addressChanged(changeRecord) {
 console.log('path: ' + changeRecord.path);
 console.log('value: ' + changeRecord.value);
}
```

Observers en arrays

Igual que los observers complejos se declaran en el array de observers, apuntado a los splices del array.
Para que se reflejen los cambios en la propiedad debemos realizarlos obligatoriamente con los métodos de polymer, push, pop, shift, unshift, y splice .
Cada cambio nos proporcionará un objeto con el siguiente formato.

- indexSplices. Contiene el conjunto de cambios ocurridos en el array.
  - index. Posición en la que el splice empieza.
  - removed. Array de elementos eliminados.
  - addedCount. Número de items insertados en el índice.
  - object: Referencia al propio array.
  - type: String literal 'splice'.

```
static get observers() {
   return ['usersAddedOrRemoved(users.splices)']
 }
usersAddedOrRemoved(changeRecord) {
  changeRecord.indexSplices.forEach(function(s) {
    s.removed.forEach(function(user) {
      console.log(user.name + ' was removed');
    });
    for (var i=0; i<s.addedCount; i++) {
      var index = s.index + i;
      var newUser = s.object[index];
      console.log('User ' + newUser.name + ' added at index ' + index);
    }
  }, this);
}
```

EVENTOS

Eventos en línea

A la hora de asociar un evento sobre un elemento de nuestro shadow DOM, solo es necesario añadir un atributo “on-” seguido del nombre del evento (on-event). Esto evita dar identificaciones a los elementos y tener que hacer búsquedas sobre estos, además de facilitar el bindeo de this.

Ya que esta asignación se hace sobre el HTML y este no diferencia entre en mayúsculas y minúsculas es conveniente que los nombres de los eventos sigan la notación por guiones (event-name) de los atributos, para evitar confusiones.

```
<dom-module id="x-custom">
   <template>
     <button on-click="handleClick">Kick Me</button>
   </template>
   <script>
     class XCustom extends Polymer.Element {
        static get is() {return 'x-custom'}
        handleClick() {
         console.log('Ow!');
       }
     }
     customElements.define(XCustom.is, XCustom);
   </script>
</dom-module>
```

Añadir listeners de manera imperativa

Listener sobre un custom-element

Los listener sobre el propio elemento se declaran usando this.addEventListener().
Lo conveniente es hacer esta declaración en el ready() o en el connectedCallback()

```
ready() {
  super.ready();
  this.addEventListener('click', this._onClick);
}

_onClick(event) {
  this._makeCoffee();
}
```

Listener sobre un elemento hijo

Lo más recomendable en este caso es usar las anotaciones en línea.
En el caso de hacerlo imperativamente es importante bindear el this, o bien con la función bind() o con una arrow-function

```
ready() {
  super.ready();
  const childElement = ...
  childElement.addEventListener('click', this._onClick.bind(this));
  childElement.addEventListener('hover', event => this._onHover(event));
}
```

Añadir listeners de manera imperativa

Listener sobre un elemento exterior

Para escuchar un evento sobre un elemento cualquiera de la página usaremos el connectedCallback() para añadir el listener y el disconnectedCallback() para eliminarlo.

Es importante eliminar estos listener para evitar memory leaks, ya que al estar hechos sobre el window o el document el garbage collector no suele eliminarlos.

```
constructor() {
 super();
 this._boundListener = this._myLocationListener.bind(this);}

connectedCallback() {
 super.connectedCallback();
 window.addEventListener('hashchange', this._boundListener);
}

disconnectedCallback() {
 super.disconnectedCallback();
 window.removeEventListener('hashchange', this._boundListener);
}
```

Eventos personalizados

Para lanzar un evento propio desde nuestro custom-element utilizaremos la función CustomEvent como constructora y el método dispatchEvent.

```
handleClick(e) {
 this.dispatchEvent(new CustomEvent('kick', {detail: {kicked: true}}));
}
```

Por defecto CustomEvent para el límite del shadow DOM. Para que nuestro evento se propague por el resto del DOM debemos crear nuestro evento con las propiedades bubbles y composed a true (por defecto estas propiedades siempre son false)

```
var event = new CustomEvent('my-event', {bubbles: true, composed: true});
```

Eventos gestuales

Para lanzar un evento propio desde nuestro custom-element utilizaremos la función CustomEvent como constructora y el método dispatchEvent.

```
<link rel="import" href="polymer/lib/mixins/gesture-event-listeners.html">

<script>
   class TestEvent extends Polymer.GestureEventListeners(Polymer.Element) {
      ...
   }
</script>
```

Para poder hacer un listener de estos eventos no podemos usar un addEventListener sin más, ya que estos eventos requieren de una configuración extra.

- Usar notación en línea

  ```
  <div id="dragme" on-track="handleTrack">Drag me!</div>
  ```

  En este caso polymer hace la configuración extra por nosotros

- Uso de los métodos Polymer.Gestures.addListener/Polymer.Gestures.removeListener

  ```
  Polymer.Gestures.addListener(this, 'track', e => this.trackHandler(e));
  ```

Tipos de eventos gestuales

Eventos soportados con una pequeña descripción de los que podemos encontrar en su event.detail.

- **down** - Cuando el dedo/ratón baja
  - x - Coordenada x
  - y - Coordenada y
  - sourceEvent - El evento original del DOM

- **up** - Cuando el dedo/ratón se levanta
  - x - Coordenada x
  - y - Coordenada y
  - sourceEvent - El evento original del DOM

- **tap** - down + up
  - x - Coordenada x
  - y - Coordenada y
  - sourceEvent - El evento original del DOM

Tipos de eventos gestuales

- **track** - Movimiento del dedo/ratón mientras está abajo
  - state - String que indica el estado del track
    - start - Lanzado cuando el track empieza (down)
    - track - Lanzado durante el trackeo
    - end - Lanzado cuando el track termina (up)
  - x - Coordenada x
  - y - Coordenada y
  - dx - Cambio de la posición horizontal desde el primer track
  - dy - Cambio de la posición vertical desde el primer track
  - ddx - Cambio de la posición horizontal desde el último track
  - ddy - Cambio de la posición vertical desde el último track
  - hover() - Función que puede ser llamada para determinar si se está pasando por un elemento concreto.

DOM TEMPLATING

“

El DOM templating provee una manera sencilla de crear un sub-árbol DOM para los componentes.

Además habilita el data-binding y el manejo de eventos de forma declarativa.

DOM template

Usando dom-module

Para definir un template usando un dom-module:

- Crear una etiqueta <dom-module> con un id que coincida con el nombre del componente.
- Añadir un elemento <template> dentro del <dom-module>
- Crear un método estático is con el nombre del elemento. El mismo del id del dom-module

```
<dom-module id="x-foo">
 <template>I am x-foo!</template>
 <script>
   class XFoo extends Polymer.Element {
     static get is() { return  'x-foo' }
    }
    customElements.define(XFoo.is, XFoo);
 </script>
</dom-module>
```

DOM template

Usando string template

Se pueden definir los templates con un método estático get template que devuelve un string.

Este método se llama cuando el elemento se instancia.

Al usar strings templates no es necesario definir el método is.

```
class MyElement extends Polymer.Element {
  static get template() {
    return `<style>:host { color: blue; }</style>
      <h2>String template</h2>
      <div>I've got a string template!</div>`
  }
}
```

Templates especiales dom-repeat

El template de repetición es un template especial que bindea un array. Crea una instancia del contenido del template por cada elemento del array. Por cada instancia se crea un scope con las siguientes propiedades:

- Item: El elemento del array usado para crear la instancia.
- Index: El indice del elemento del array (Cambia si el array es ordenado o filtrado)

Hay dos formas de usar el dom-repeat:

- Dentro de un elemento o un template de Polymer

```
<template is="dom-repeat" items="{{items}}">
  ...
</template>
```

- Fuera de un elemento o un template de Polymer

```
<dom-repeat items="{{items}}">
  <template>
    ...
  </template>
</dom-repeat>
```

Templates especiales dom-repeat

Manejo de eventos

Cuando se declara un manejador de eventos de manera declarativa dentro de un dom-repeat, el templete añade una propiedad model por cada evento enviado a un listener. El objeto model contiene los datos usados para la generación de la instancia, por ejemplo los datos referentes al elemento estarán en model.item

```
<template is="dom-repeat" id="menu" items="{{menuItems}}">
 <div>
   <span>{{item.name}}</span>
   <span>{{item.ordered}}</span>
   <button on-click="order">Order</button>
 </div>
</template>

...

order(e) {
  e.model.set('item.ordered', e.model.item.ordered+1);
}
```

Templates especiales dom-repeat

Filtrado y ordenación de listas

Para filtrar u ordenar los elementos mostrados de una listas, se puede añadir las propiedades filter y sort al dom-repeat.

- filter: Especifica una función de callback que recibe el item como argumento y devuelve true para mostrar el elemento y false si hay que omitirlo.
- sort: Especifica una función de comparación siguiendo el estándar de la función sort del API de array.

Por defecto estas funciones se ejecutan cuando un observable cambia o cambia la función de filter o sort.

Cada vez que se ejecuta una de estas funciones se produce una llamada a la función render de dom-repeat.

```
<template is="dom-repeat" id="menu" items="{{employees}}" filter="{{isEngineer}}">
  ...
</template>

...

isEngineer(item) {
  return item.type == 'engineer' || item.manager.type == 'engineer';
}
```

Templates especiales dom-if

El template condicional inserta su contenido en el DOM solo cuando la propiedad if es true.

Si la condición se vuelve falsa, los elementos del template condicional se ocultan, pero no se eliminan del DOM.

Hay dos formas de usar el dom-if:

- Dentro de un elemento o un template de Polymer

```
<template is="dom-if" if="{{condition}}">
  ...
</template>
```

- Fuera de un elemento o un template de Polymer

```
<dom-repeat if="{{condition}}">
  <template>
    ...
  </template>
</dom-repeat>
```

Templates especiales dom-bind

El data binding de Polymer funciona solo en templates gestionados por Polymer (polymer-elements, dom-repeat, dom-if), pero no para los elementos alojados en el documento principal.

Para usar el bindeo de datos sin definir un custom element, podemos usar el elemento <dom-bind>. Este template inserta su contenido automáticamente en el documento principal permitiendo sincronización de los datos en ese ámbito.

```
<dom-bind>
 <template>

   <!-- Note the data property which gets sets below -->
   <template is="dom-repeat" items="{{data}}">
     <div>{{item.name}}: {{item.price}}</div>
   </template>

 </template>
</dom-bind>
<script>
 var autobind = document.querySelector('dom-bind');

 // set data property on dom-bind
 autobind.data = [
   { name: 'book', price: '$5.00'},
   { name: 'pencil', price: '$1.00'},
   { name: 'flux capacitor', price: '$8,000,000.00'}
 ];
</script>
```

Mapa de nodos estático

Polymer construye un mapa de los nodos con ID cuando se inicializa el shadow DOM, para facilitar el acceso a los nodos que puedan ser más utilizados sin tener que realizar búsquedas manuales. Cada nodo definido con un id es almacenado en el objeto this.$ con el id como hash.

Los nodos creados dinámicamente o dentro de un un dom-repeat o un dom-if no se añadirán al objeto this.$

```
<dom-module id="x-custom">
  <template>
   Hello World from <span id="name"></span>!
 </template>
 <script>
   class MyElement extends Polymer.Element {
     static get is() { return  'x-custom' }
     ready() {
       super.ready();
       this.$.name.textContent = this.tagName;
     }
   }
 </script>
</dom-module>
```

ESTILOS

Estilos sobre elementos

Una de las mayores ventajas que nos aporta el shadow DOM es la de encapsular nuestro HTML y CSS de manera que los estilos no interfieran con el resto de los estilos de la página.

```
<dom-module id='x-foo'>
 <template>
   <!-- Encapsulated, element-level stylesheet -->
   <style>
     p {
       color: green;
     }
     .myclass {
       color: red;
     }
   </style>
   <p>I'm a shadow DOM child element of x-foo.</p>
   <p class="myclass">So am I.</p>
 </template>
 ...
</dom-module>
```

```
<link rel="import" href="x-foo.html">
<style>
 .myclass {
   color: blue;
 }
</style>
<x-foo></x-foo>
<p class="myclass">I have nothing to do with x-foo. Because of encapsulation,
x-foo's styles won't leak to me.</p>
```

Estilos sobre elementos

Herencia de los estilos del DOM

Al usar el elemento en un documento HTML heredamos los estilos del elemento padre. Aunque los estilos declarados en el shadow DOM sobreescribiran los declarados fuera

```
<dom-module id='x-foo'>
 <template>
   <!-- Encapsulated, element-level stylesheet -->
   <style>
     p {
       color: green;
     }
   </style>
   <p>I'm green.</p>
 </template>
</dom-module>
```

```
<link rel="import" href="x-foo.html">
<!-- Document-level stylesheet -->
<style>
 p {
   font-family: sans-serif;
   color: blue;
 }
</style>
<p>I'm blue.</p>
<p><x-foo></x-foo></p>
```

Estilos sobre elementos

Estilos sobre el host

El elemento sobre el que se inserta el shadow DOM se le llama host, y este se le pueden aplicar estilos utilizando el selector :host.

```
<dom-module id="x-foo">
 <template>
   <style>
     :host { font-family: sans-serif; }
     :host(.blue) {color: blue;}
     :host([selected]) {color: red;}
     :host(:hover) {color: green;}
   </style>
   <p>Hi, from x-foo!</p>
 </template>
</dom-module>
```

```
<link rel="import" href="x-foo.html">
<x-foo class="blue"></x-foo>
<x-foo class="red"></x-foo>
```

Los estilos del host también se pueden cambiar desde fuera

```
x-foo {
  background-color: blue;
}
```

Estilos sobre elementos

Estilos en nodos distribuidos (slot content)

A través de selector ::slotted podemos asignar estilos a los elementos que se introduzcan por el light-dom.

```
<dom-module id="x-foo">
 <template>
   <style>
     h1 ::slotted(h1) {
       font-family: sans-serif;
       color: green;
     }
     p ::slotted(p) {
       font-family: sans-serif;
       color: blue;
     }
   </style>
   <h1><slot name='heading1'></slot></h1>
   <p><slot name='para'></slot></p>
 </template>
 ...
</dom-module>
```

```
<link rel="import" href="x-foo.html">

<x-foo>
 <h1 slot="heading1">Heading 1. I'm green.</h1>
 <p slot="para">Paragraph text. I'm blue.</p>
</x-foo>
```

Compartir estilos entre componentes

La mejor forma de compartir estilos entre diferentes módulos es encapsulando los estilos en un módulo y compartiéndolo entre ellos

```
<dom-module id='my-colors'>
   <template>
       <style>
       p.red {
           color: red;
       }
       p.green {
           color: green;
       }
       p.blue {
           color: blue;
       }
       </style>
   </template>
</dom-module>
```

```
<link rel="import" href="my-colors.html">
<dom-module id="x-foo">
 <template>
   <!-- Include the imported styles from my-colors -->
   <style include="my-colors"></style>
   <p class="red">I wanna be red</p>
   <p class="green">I wanna be green</p>
   <p class="blue">I wanna be blue</p>
 </template>
 ...
</dom-module>
```

Theming y custom properties

Los estilos de los elementos del shadow dom de un componente no se pueden cambiar desde fuera a excepción de los estilos que se hereden de la cascada de CSS como las fuentes y el color.

Para permitir a los usuario personalizar los componentes podemos exponer ciertas propiedades usando las custom properties y los mixins de custom properties.

Las custom properties son una especie de variables que se pueden modificar en las reglas de CSS.

```
:host {
 background-color: var(--my-theme-color);
}
```

Estas propiedades se pueden reasignar desde fuera a alto nivel.

```
html {
 --my-theme-color: red;
}
```

Las custom properties pueden tener un valor por defecto. Este valor puede ser incluso otra custom properties.

```
:host {
   background-color: var(--my-theme-color, blue);
   background-color: var(--my-theme-color, var(--another-theme-color, blue));
}
```

Theming y custom properties

Custom properties mixins

Los mixins de custom properties son una característica a alto nivel de la especificación. Básicamente, el mixin es una variable que contiene múltiples propiedades.

```
html {
 --my-custom-mixin: {
   color: white;
   background-color: blue;
 }
}
```

Un componente puede importar un mixin usando la regla @apply.

```
:host {
 @apply --my-custom-mixin;
}
```

Custom properties API

Los valores de las custom properties son aplicados y evaluados en el momento en el que el componente se crea. Estos valores pueden ser re-evaluados debido a cambios dinámicos que ocurren en nuestra aplicación, para ello invocamos el método UpdateStyles.

UpdateStyles recibe un objeto con los pares/valor con las custom properties que se van a modificar.

```
<dom-module id="x-custom">
    <template>
        <style>
            :host {
                --my-toolbar-color: red;
            }
        </style>
        <my-toolbar>My awesome app</my-toolbar>
        <button on-tap="changeTheme">Change theme</button>
    </template>
    <script>
        class XCustom extends Polymer.Element {
            static get is() { return "x-custom"; }
            static get changeTheme() {
                return function() {
                    this.updateStyles({
                        '--my-toolbar-color': 'blue',
                    });
                }
            }
        }
        customElements.define(XCustom.is, XCustom);
    </script>
</dom-module>
```

MIXINS

Las clases de ES6 no permiten la herencia múltiple, esto dificulta el compartir código entre elementos no relacionados

Los mixins nos permiten compartir código entre elementos sin tener una superclase común. Básicamente son un función generadora de clases

Uso de los mixins

Lo primero será añadir el mixin a nuestro elemento.

```
class MyElement extends MyMixin(Polymer.Element) {
 static get is() { return 'my-element' }
}
```

La jerarquia de la herencia sería la siguiente:
MyElement <= MyMixin <= Polymer.Element.

Los mixins al ser factorías de clases que reciben una superclase nos permiten simular una especie de herencia múltiple.

```
MyCompositeMixin = (base) => class extends MyMixin2(MyMixin1(base)) {
 ...
}
```

Definición de los mixins

```
MyMixin = superClass =>  {
 return class extends superClass {
   constructor() {
     super();
     this.addEventListener('keypress', e => this.handlePress(e));
   }

   static get properties() {
     return {
       bar: {
         type: Object
       }
     };
   }

   static get observers() {
     return [ '_barChanged(bar.*)' ];
   }

   _barChanged(bar) { ... }

   handlePress(e) { console.log('key pressed: ' + e.charCode); }
 }
}
```

POLYMER CLI

“

Polymer CLI es la herramienta de línea de comandos para proyectos Polymer. Incluye tareas de construcción, generación de componentes y apps, linter, servidor de desarrollo y test runner.

Instalación del Polymer CLI

1. Asegurarse de tener instalada una versión de node compatible con Polymer CLI (mayor de la 6).
   node --version

2. Actualizar npm.
   npm install npm@latest -g

3. Asegurarse de que git esté instalado.
   git --version

   Si no lo esta podemos descargarlo en su página de descarga.

4. Instalar la última versión de Bower.
   npm install -g bower

5. Instalar Polymer CLI.
   npm install -g polymer-cli

Comandos Polymer CLI

Polymer build

Genera una versión de producción de la aplicación. Este proceso incluye el minimizado del HTML, CSS y JS de la aplicación y sus dependencias, y la generación de un service worker para pre-cachear las dependencias.

El proceso de construcción del CLI está diseñado siguiendo el patrón PRPL.

Para asegurar que la construcción de la app es la correcta, se crea un archivo polymer.json en el primer nivel de la estructura de archivos, en el que están definidos los criterios de la construcción.

Además se pueden añadir flags a la construcción de manera que podamos obtener una app más personalizada.

Ajustes preestablecidos del proceso de construcción:

- **--add-service-worker:** Genera un service worker para la aplicación que cachea los ficheros y las dependencias de la aplicación en el cliente.
  El service worker se puede personalizar modificando el archivo sw-precache-config.js del directorio raíz.
- **--bundle:** Por defecto el proceso de construcción no agrupa los fragmentos. Esto es óptimo para servidores con HTTP/2.
  Usando este flag tendremos una construcción agrupada lo que reduce el número de peticiones

Comandos Polymer CLI
(polymer build)

- **--css-minify:** Minimiza el css interno y externo
- **--entrypoint:** Cambia el punto de entrada
- **--html-minify:** Minimiza el html eliminando comentarios y espacios en blanco
- **--insert-prefetch-link:** Inserta elementos de enlace de captación previa en tus fragmentos para que todas las dependencias sean captadas inmediatamente. Agregue la recuperación previa de la dependencia insertando las etiquetas <link rel = "prefetch"> en entrypoint y <link rel = "import"> en fragmentos y shell para todas las dependencias.
- **--js-compile:** Usa Babel para transpilar el javascript en ES6 a ES5 para navegadores antiguos.
- **--js-minify:** Minimiza el js interno y externo
- **--shell:** Fichero que contiene el código común a la aplicación
- **--fragment:** Agrupación de las dependencias dinámicas. Es un array de archivos HTML que no están enlazados estáticamente a la aplicación (la carga se hace bajo demanda con importHref)

Comandos Polymer CLI
(polymer build)

Construcciones preconfiguradas.

La herramienta provee las configuraciones más comunes agrupadas.

- **es5-bundled:** --js-minify --js-compile --css-minify --bundled --add-service-worker --add-push-manifest --insert-prefetch-links
- **es6-bundled:** --js-minify --css-minify --html-minify --bundled --add-service-worker --add-push-manifest --insert-prefetch-links
- **es6-unbundled:** --js-minify --css-minify --html-minify --add-service-worker --add-push-manifest --insert-prefetch-links

Un ejemplo sería: polymer build --preset es5-bundled

Comandos Polymer CLI

Polymer init

Inicia un proyecto Polimer con una de las plantillas pre-establecidas.

Al ejecutarse el comando se mostrará una lista de las plantillas disponibles, o bien si se conoce la plantilla se puede pasar como argumento.

Las plantillas pre-establecidas disponibles son:

- Crear un proyecto de elemento con la CLI de Polymer
- Cree un proyecto de aplicación con Polymer CLI
- Estudio de caso de la aplicación Polymer Shop

Polymer install

Instala las dependencias definidas en el bower.json.

El flag --variants permite instalar variates de dependencia.

El flag --offline recupera las dependencias de la caché en lugar de la red. Si los componentes no están en la memoria caché la instalación fallará.

Comandos Polymer CLI

Polymer lint

Analiza el proyecto para detectar errores de sintaxis, importaciones faltantes, expresiones de enlace de datos incorrectas y más.

Se usa de la siguiente manera: polymer lint --rules=polymer-2

Se puede configurar el linter añadiendo nuestras propias reglas en el package.json de manera que otras herramientas puedan usar las mismas reglas.

```
{
  "lint": {
      "rules": ["polymer-2"],
      "ignoreWarnings": []
  }
}
```

Polymer serve

Arranca un servidor local

polymer serve

Comandos Polymer CLI
(polymer serve)

Server options

- **--port:** polymer serve --port 3000
- **--hostname:** polymer serve --hostname test
- **--open:** Permite abrir una página diferente a la por defecto, y en otro navegador que no sea el predeterminado polymer serve --open app.html --browser Safari
- **--compile:** El servidor automáticamente transpila de ES6 a ES5.
  polymer serve --compile always
  polymer serve --compile never
  polymer serve --compile auto

Polymer test

Ejecuta los test de los componentes de la aplicación: polymer test

POLYMER 3

HTML Imports → Módulos de ES6

Desde el principio, Polymer ha utilizado HTML Imports para cargar las dependencias.

Sin embargo, HTML Imports no han sido acogidos complementamente por los comités de estandarización y otros navegadores.

Los módulos ES6 permiten a los archivos JavaScript importar otros archivos, haciendo que sean cargados y ejecutados por el navegador. El comportamiento de carga de los módulos ES6 es casi idéntico a los HTML Imports.

- Son un mecanismo nativo de la web.
- Carga transitiva de dependencias con evaluación ordenada.
- Deduplicación de dependencias mediante la URL.

Evidentemente falta la característica de la carga y el análisis nativo del código HTML importado.

```
// Antes para exportar
window.MiElemento = // ...
// Ahora
export const MiElemento = // …
```

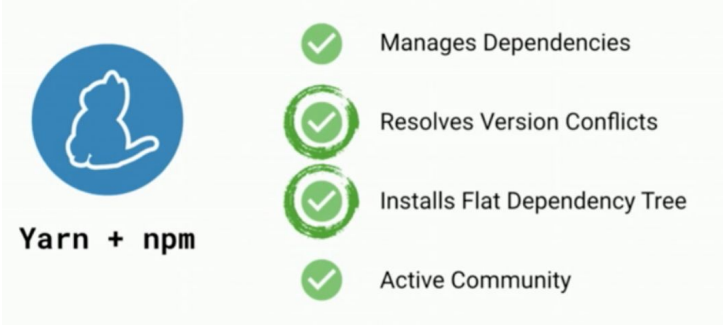

```
// Antes para importar
<link rel="import" href="../@polymer/polymer/polymer-element.html" />
// Ahora
import { Element } from '../@polymer/polymer/polymer-element'
```

Bower ➞ npm

Al igual que los HTML Imports, también hemos utilizado Bower durante mucho tiempo. El árbol plano de dependencias de Bower es ideal para proyectos de front-end. Pero Bower nunca ha logrado una adopción tan extensa como npm, y aunque aún lo están manteniendo, ya no está siendo desarrollando activamente.

Migrar a npm hará que los paquetes de Polymer estén disponibles para los millones de usuarios con los que cuenta npm y permitirá que estos puedan utilizar más fácilmente otros paquetes del masivo ecosistema de npm.

El cliente npm de yarn proporciona soporte para instalaciones planas, lo que soluciona nuestro principal problema para utilizar npm.



Qué hay de nuevo en Polymer 3.0

Polymer-modulizer

Herramienta que se encarga del trabajo mecánico de pasar los custom elements que usan HTML Imports a Javascript Modules.

No funciona con Polymer 1.x ya que no usa las clases de ES6 para definir los componentes.

HTML templates dentro de Javascript

Polymer 3 usa los ES6 template String, permitiendo escribir cadenas con templates en varias líneas y pudiendo interpolar expresiones de manera nativa.

Los templates HTML sin embargo se mantienen en Polymer 3 por motivos de compatibilidad hacia atrás, de modo que haya un camino suavizado de migración entre componentes 2.x a 3.x, facilitado también por el mencionado polymer-modulizer.

LitElement

Esta es una de las novedades de Polymer 3.0 más importantes en lo que respecta al rendimiento. Ya que ahora la definición de los templates se realiza mediante Javascript, el equipo de Polymer ha trabajado en una nueva manera de definir estos templates, que aproveche en mayor medida las capacidades nativas de la plataforma web.

Qué hay de nuevo en Polymer 3.0

Nuevo catálogo de componentes

Material-Components, que es una hornada de componentes Material Design que se han presentado de manera conjunta para usar en las tres plataformas principales: Android nativo, iOS nativo y Web.

Este nuevo catálogo de componentes, que sustituye a los conocidos "Paper Elements" usa las guías de diseño de Material Design 2.

PWA Starter Kit

Básicamente nos ofrece un template nuevo para comenzar el desarrollo de una Progressive Web App, basado en Polymer 3.0 y LitElement. El PWA Starter Kit viene con diversas novedades, pero que básicamente se pueden resumir en:

- Existen diversas plantillas de aplicaciones, con unas y otras variantes, adaptables a un público mayor.
- Usa componentes nuevos, basados en el mencionado Material Web Components
- Ofrece comandos para test, build y otras operaciones para desarrollo.
- Usa un patrón unidireccional de data-binding, montado encima de la librería Redux.