Usando custom elements

Um dos principais recursos do padrão de Web Components é a capacidade de criar elementos personalizados que encapsulam sua funcionalidade em uma página HTML, em vez de ter que se contentar com um lote longo e aninhado de elementos que, juntos, fornecem um recurso de página personalizada. Este artigo apresenta o uso da API de Custom Elements.

> **Nota**: Custom elements são suportados por padrão no Firefox, Chrome e Edge (76). Opera e Safari até agora suportam apenas custom elements autônomos.

Visão de alto nível

O controlador de custom elements em um documento da web é o objeto `CustomElementRegistry` — este objeto permite que você registre um custom element na página, retorne informações sobre quais custom elements estão registrados, etc..

Para registar um custom element na página, use o método `CustomElementRegistry.define()`. Isso leva como argumentos:

- Um `DOMString` que representa o nome que você está dando ao elemento. Observe que os nomes dos custom elements requerem o uso de um traço (kebab-case); não podem ser palavras isoladas.
- Um objeto de classe que define o comportamento do elemento.
- Opcionalmente, um objeto de opções contendo uma propriedade `extends`, que especifica o elemento integrado do qual seu elemento herda, se houver (relevante apenas para elementos integrados personalizados; consulte a definição abaixo).

Então, por exemplo, podemos definir um custom element word-count `(contagem-palavras)` assim:

```
customElements.define('word-count', WordCount, { extends: 'p' });
```

O elemento é chamado de `word-count`, seu objeto de classe é `WordCount`, e estende o elemento `<p>`.

O objeto de classe de um custom element é escrito usando a sintaxe de classe ES 2015. Por exemplo, `WordCount` é estruturado assim::

```
class WordCount extends HTMLParagraphElement {
  constructor() {
    // Sempre chame super primeiro no construtor
    super();

    // Funcionalidade do elemento escrita aqui

    ...
  }
}
```

Este é apenas um exemplo simples, mas você pode fazer mais aqui. É possível definir retornos de chamada de ciclo de vida específicos dentro da classe, que são executados em pontos específicos do ciclo de vida do elemento. Por exemplo, `connectedCallback` é invocado cada vez que o custom element é anexado a um elemento conectado ao

documento, enquanto `attributeChangedCallback` é invocado quando um dos atributos do elemento customizado é adicionado, removido ou alterado.

Você aprenderá mais sobre eles na seção Using the lifecycle callbacks abaixo.

Existem dois tipos de custom elements:

- **Autonomous custom elements** são autonômos — eles não herdam de elementos HTML padrão. Você os usa em uma página, literalmente escrevendo-os como um elemento HTML. Por exemplo `<popup-info>`, ou `document.createElement("popup-info")`.
- **Customized built-in elements** herdam de elementos HTML básicos. Para criar um deles, você deve especificar qual elemento eles estendem (como implícito nos exemplos acima), e eles são usados escrevendo o elemento básico, mas especificando o nome do elemento personalizado no atributo `is` (ou propriedade). Por exemplo `<p is="word-count">`, ou `document.createElement("p", { is: "word-count" })`.

Trabalhando com alguns exemplos simples

Neste ponto, vamos examinar mais alguns exemplos simples para mostrar como os custom elements são criados com mais detalhes.

Custom elements autônomos

Vamos dar uma olhada em um exemplo de um custom element autônomo — `<popup-info-box>` (veja um exemplo ao vivo). Isso pega um imagem de ícone e uma sequência de texto e incorpora o ícone na página. Quando o ícone está em foco, ele exibe o texto em uma caixa pop-up de informações para fornecer mais informações no contexto.

Para começar, o arquivo JavaScript define uma classe chamada `PopUpInfo`, que estende a classe genérica `HTMLElement`.

```
class PopUpInfo extends HTMLElement {
  constructor() {
```

```
    // Sempre chame super primeiro no construtor
    super();

    // escreva a funcionalidade do elemento aqui

    ...
  }
}
```

O trecho de código anterior contém a definição do `constructor()` da classe, que sempre começa chamando `super()` para que a cadeia de protótipo correta seja estabelecida.

Dentro do construtor, definimos toda a funcionalidade que o elemento terá quando uma instância dele for instanciada. Neste caso, anexamos uma shadow root ao custom element, usamos alguma manipulação de DOM para criar a estrutura de shadow DOM interna do elemento - que é então anexada à shadow root - e, finalmente, anexamos algum CSS à shadow root para estilizá-la.

```
// Create a shadow root
this.attachShadow({mode: 'open'}); // sets and returns 'this.shadowRoot'

// Create (nested) span elements
const wrapper = document.createElement('span');
wrapper.setAttribute('class','wrapper');
const icon = wrapper.appendChild(document.createElement('span'));
icon.setAttribute('class','icon');
icon.setAttribute('tabindex', 0);
// Insert icon from defined attribute or default icon
const img = icon.appendChild(document.createElement('img'));
img.src = this.hasAttribute('img') ? this.getAttribute('img') :
'img/default.png';

const info = wrapper.appendChild(document.createElement('span'));
info.setAttribute('class','info');
// Take attribute content and put it inside the info span
info.textContent = this.getAttribute('data-text');
```

```
// Create some CSS to apply to the shadow dom
const style = document.createElement('style');
style.textContent = '.wrapper {' +
// CSS truncated for brevity

// attach the created elements to the shadow DOM
this.shadowRoot.append(style,wrapper);
```

Por fim, registramos nosso custom element no `CustomElementRegistry` usando o método `define()` mencionado anteriormente — nos parâmetros especificamos o nome do elemento e, em seguida, o nome da classe que define sua funcionalidade:

```
customElements.define('popup-info', PopUpInfo);
```

Agora está disponível para uso em nossa página. Em nosso HTML, nós o usamos assim:

```
<popup-info img="img/alt.png" data-text="Your card validation code (CVC)
  is an extra security feature — it is the last 3 or 4 numbers on the
  back of your card."></popup-info>
```

> **Note**: Você pode ver o código-fonte JavaScript completo aqui.

Estilos internos vs. externos

No exemplo acima, aplicamos o estilo ao Shadow DOM usando um elemento `<style>`, mas é perfeitamente possível fazer isso referenciando uma folha de estilo externa de um elemento `<link>` em vez disso.

Por exemplo, dê uma olhada neste código de nosso exemplo popup-info-box-external-stylesheet (veja o código-fonte):

```
// Aplicar estilos externos ao shadow dom
const linkElem = document.createElement('link');
linkElem.setAttribute('rel', 'stylesheet');
linkElem.setAttribute('href', 'style.css');

// Anexe o elemento criado ao shadow dom
shadow.appendChild(linkElem);
```

Observe que os elementos `<link>` não bloqueiam a pintura do shadow root, portanto, pode haver um flash de conteúdo não estilizado (FOUC) enquanto a folha de estilo é carregada.

Muitos navegadores modernos implementam uma otimização para tags `<style>` clonadas de um nó comum ou que tenham texto idêntico, para permitir que compartilhem uma única folha de estilo de apoio. Com essa otimização, o desempenho dos estilos externo e interno deve ser semelhante.

Customized built-in elements

Agora vamos dar uma olhada em outro exemplo de custom element integrado — expanding-list (ver ao vivo também). Isso transforma qualquer lista não ordenada em um menu de expansão/recolhimento.

Em primeiro lugar, definimos a classe do nosso elemento, da mesma maneira que antes:

```
class ExpandingList extends HTMLUListElement {
  constructor() {
    // Sempre chame super primeiro no construtor
    super();

    // escreva a funcionalidade do elemento aqui

    ...
  }
}
```

Não explicaremos a funcionalidade do elemento em detalhes aqui, mas você pode descobrir como ele funciona verificando o código-fonte. A única diferença real aqui é que nosso elemento está estendendo a interface `HTMLUListElement` (en-US), e não `HTMLElement`. Portanto, ele tem todas as características de um elemento `<ul>` com a funcionalidade que definimos construída no topo, ao invés de ser um elemento autônomo. Isso é o que o torna um elemento integrado personalizado, em vez de um elemento autônomo.

Em seguida, registramos o elemento usando o método `define()` como antes, exceto que, desta vez, ele também inclui um objeto de opções que detalha de qual elemento nosso elemento personalizado herda:

```
customElements.define('expanding-list', ExpandingList, { extends: "ul" });
```

Usar o elemento integrado em um documento da web também parece um pouco diferente:

```
<ul is="expanding-list">

  ...

</ul>
```

Você usa um elemento `<ul>` normalmente, mas especifica o nome do elemento personalizado dentro do atributo `is`.

> **Nota**: Novamente, você pode ver o código-fonte JavaScript completo aqui.

Usando os callbacks do ciclo de vida

Você pode definir vários retornos de chamada diferentes dentro da definição de classe de um custom element, que disparam em diferentes pontos do ciclo de vida do elemento:

- `connectedCallback` : Chamado sempre que o custom element é anexado a um elemento conectado ao documento. Isso acontecerá sempre que o nó for movido e pode acontecer antes que o conteúdo do elemento tenha sido totalmente analisado.

> **Nota:** `connectedCallback` pode ser chamado assim que seu elemento não estiver mais conectado, use `Node.isConnected` para ter certeza.

- `disconnectedCallback` : Invocado sempre que o custom element é desconectado do documento DOM.
- `adoptedCallback` : Invocado sempre que o custom element é movido para um novo documento.
- `attributeChangedCallback` : Invocado sempre que um dos atributos do custom element é adicionado, removido ou alterado. Os atributos a serem observados na mudança são especificados em um método estático `observedAttributes`

Vejamos um exemplo em uso. O código abaixo é retirado do exemplo life-cycle-callbacks (ver rodando ao vivo). Este é um exemplo trivial que simplesmente gera um quadrado colorido de tamanho fixo na página. O custom element tem a seguinte aparência:

```
<custom-square l="100" c="red"></custom-square>
```

O construtor da classe é realmente simples - aqui anexamos um shadow DOM ao elemento e, em seguida, anexamos os elementos vazios `<div>` e `<style>` ao shadow root:

```
const shadow = this.attachShadow({mode: 'open'});

const div = document.createElement('div');
const style = document.createElement('style');
shadow.appendChild(style);
shadow.appendChild(div);
```

A função chave neste exemplo é `updateStyle()` — isso pega um elemento, pega seu shadow root, encontra seu elemento `<style>`, e adiciona `width`, `height`, e `background-color` para o estilo.

```
function updateStyle(elem) {
  const shadow = elem.shadowRoot;
  shadow.querySelector('style').textContent = `
    div {
      width: ${elem.getAttribute('l')}px;
      height: ${elem.getAttribute('l')}px;
      background-color: ${elem.getAttribute('c')};
    }
  `;
}
```

As atualizações reais são todas tratadas pelos retornos de chamada do ciclo de vida, que são colocados dentro da definição da classe como métodos. O `connectedCallback()` é executado sempre que o elemento é adicionado ao DOM - aqui, executamos a função `updateStyle()` para garantir que o quadrado seja estilizado conforme definido em seus atributos:

```
connectedCallback() {
  console.log('Custom square element added to page.');
  updateStyle(this);
}
```

Os retornos de chamada `disconnectedCallback()` e `adoptedCallback()` registram mensagens simples no console para nos informar quando o elemento é removido do DOM ou movido para uma página diferente:

```
disconnectedCallback() {
  console.log('Custom square element removed from page.');
}

adoptedCallback() {
```

```
    console.log('Custom square element moved to new page.');
  }
```

O `attributeChangedCallback()` é executado sempre que um dos atributos do elemento é alterado de alguma forma. Como você pode ver por suas propriedades, é possível atuar sobre os atributos individualmente, observando seus nomes e antigos e novos valores de atributos. Neste caso, entretanto, estamos apenas executando a função `updateStyle()` novamente para garantir que o estilo do quadrado seja atualizado de acordo com os novos valores:

```
attributeChangedCallback(name, oldValue, newValue) {
  console.log('Custom square element attributes changed.');
  updateStyle(this);
}
```

Observe que, para fazer com que o retorno de chamada `attributeChangedCallback()` seja acionado quando um atributo for alterado, você deve observar os atributos. Isso é feito especificando um método `static get observedAttributes()` dentro da classe de custom element - isso deve `retornar` um array contendo os nomes dos atributos que você deseja observar:

```
static get observedAttributes() { return ['c', 'l']; }
```

Isso é colocado bem no topo do construtor, em nosso exemplo.

> **Nota**: Encontre o código-fonte JavaScript completo aqui.

Polyfills vs. classes

Polyfills de Custom Element podem corrigir construtores nativos, como `HTMLElement` e outros, e retornar uma instância diferente daquela recém-criada.

Se você precisar de um `constructor` e uma chamada de `super` obrigatória, lembre-se de passar argumentos opcionais e retornar o resultado de tal chamada de `super`.

```
class CustomElement extends HTMLElement {
  constructor(...args) {
    const self = super(...args);
    // self functionality written in here
    // self.addEventListener(...)
    // return the right context
    return self;
  }
}
```

Se você não precisa realizar nenhuma operação no construtor, você pode simplesmente omiti-lo para que seu comportamento nativo (veja a seguir) seja preservado.

```
constructor(...args) { return super(...args); }
```

Transpiladores vs. classes

Observe que as classes ES2015 não podem ser transpiladas de forma confiável em Babel 6 ou TypeScript visando navegadores legados. Você pode usar o Babel 7 ou o babel-plugin-transform-builtin-classes para Babel 6, e target ES2015 em TypeScript em vez do legado..

Bibliotecas

Existem várias bibliotecas que são construídas em Web Components com o objetivo de aumentar o nível de abstração ao criar elementos personalizados. Algumas dessas bibliotecas são snuggsi ツ , X-Tag , Slim.js , LitElement , Smart , Stencil e hyperHTML-Element .

Last modified: 8 de abr. de 2022, by MDN contributors