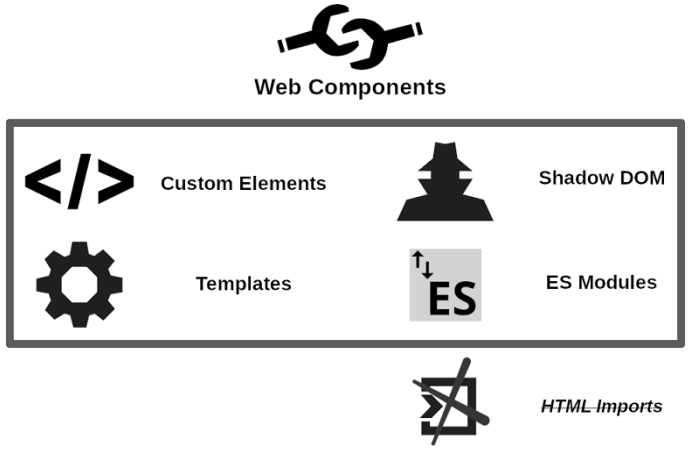
Chapitre 4
Faire bon usage des Web Components

1. Introduction



Standards utilisés pour la création de Web Components

Après plus de trente ans d'existence, et une décennie de débats et expérimentations, la plateforme web permet enfin de réaliser de véritables composants graphiques pleinement réutilisables.

Ces "Web Components" ont ces dernières années fait naître beaucoup d'espoirs et de spéculations. La grande majorité d'entre elles passe cependant à côté de la véritable nature de cette nouvelle technique.

Répondant à des problématiques précises, les Web Components complètent (et parfois remplacent) déjà plusieurs bibliothèques pour tout un ensemble de cas d'usages. Nous sommes cependant très loin de voir les Web Components prendre le pas sur les frameworks et bibliothèques de développement applicatif (si tant est que cela soit possible, voire souhaitable).

De nombreuses solutions furent créées bien avant l'arrivée des Web Components. Adapter ces solutions à des standards qui n'existaient pas au moment de leur conception demande un travail conséquent. Dans certains cas, cette adaptation peut même s'avérer impossible.

Remarque

React, notamment, repose sur un type de composants très éloignés des Web Components (voir le chapitre Adopter une approche par composant, section Liens avec une approche fonctionnelle). Il est donc possible, voire probable, que cette bibliothèque ne soit jamais modifiée au point d'associer ces deux concepts.

Dans ce chapitre, nous allons donc nous consacrer exclusivement aux Web Components eux-mêmes, afin de déterminer comment ils peuvent et doivent être développés.

Pour cela, nous utiliserons trois ensembles de techniques :

– les éléments personnalisés (*Custom Elements*), pour définir de nouveaux éléments HTML qui constitueront la base de nos Web Components ;
– des arbres fantômes (*Shadow DOM*), pour isoler le contenu d'un Web Component du contexte dans lequel il est employé ;
– et enfin, des templates HTML (*HTML Templates*), pour définir un contenu pouvant être dynamiquement affiché.

Techniques auxquelles nous associerons les bonnes pratiques à respecter, ainsi que des évolutions récentes du Web susceptibles de nous aider à concevoir des composants modernes et pérennes.

2. Étendre le vocabulaire HTML

Avant toute chose, créer des Web Components consiste à inventer de "nouveaux mots" pour HTML. Le standard de ce langage définit déjà un grand nombre d'éléments, mais tous ne peuvent répondre à nos besoin. Nous allons donc créer de nouveaux éléments personnalisés, c'est-à-dire des *Custom Elements*, pour reprendre l'expression définie dans le standard HTML.

L'ensemble de l'interface de développement HTML pour les éléments personnalisés est disponible via le seul objet `window.customElements`, instance de `CustomElementRegistry`, et n'est constitué que des quatre méthodes suivantes :

– `customElements.define()`, pour définir un élément personnalisé
– `customElements.get()`, pour renvoyer le constructeur d'un élément personnalisé si celui-ci a déjà été défini
– `customElements.upgrade()`, pour "mettre à jour" un élément personnalisé déconnecté du DOM
– `customElements.whenDefine()`, pour détecter quand un élément personnalisé donné est défini

Leur utilisation comporte cependant plusieurs particularités importantes, ainsi que des pièges à éviter.

2.1 Définir un élément personnalisé

Lorsque nous évoquons les spécifications Custom Elements, nous pensons avant tout à la définition de nouveaux éléments HTML non standardisés. Ce premier type d'élément personnalisé est dit "autonome" (*Autonomous Custom Element*).

Tout élément personnalisé autonome précédemment défini pour un nom donné (par exemple, `hello-world`) est utilisable comme n'importe quel autre élément HTML :

```
<hello-world></hello-world>
```

Le standard HTML définit également une fonctionnalité supplémentaire, permettant qu'un élément personnalisé étende un élément HTML standard. Les éléments ainsi obtenus sont appelés "élément intégrés personnalisés" (*customized build-in element*). Nous pourrons les utiliser dans un document HTML via l'attribut `is`, comme dans l'exemple suivant :

```
<p is="hello-world"></p>
```

Avant de pouvoir utiliser de tels éléments personnalisés, nous devons bien entendu les définir. Pour cela, nous devons leur attribuer à chacun :

– un sélecteur de l'élément (autrement dit, un nom)

– une classe JavaScript

– un ensemble d'options

Ces trois caractéristiques correspondent aux trois arguments de la méthode `customElements.define()`.

Pour définir l'élément personnalisé autonome `hello-world` présenté précédemment, il suffit donc, après avoir défini la classe `HelloWorldComponent`, de faire appel à cette méthode comme dans l'exemple suivant.

```
customElements.define("hello-world", HelloWorldComponent);
```

Le dernier argument, optionnel, représente sous forme d'objet l'ensemble des options contrôlant cette définition. Pour l'heure, le standard HTML ne prévoit cependant qu'une seule et unique option `extends`, permettant d'indiquer l'élément père, afin de définir un élément intégré personnalisé.

Nous avons donc défini un élément personnalisé autonome en omettant de spécifier `extends`. Pour définir l'élément intégré personnalisé étendant `<p>` précédent, il suffit donc de spécifier "`p`" comme valeur pour cette option.

```
customElements.define(
 "hello-world",
 HelloWorldParagraphComponent,
 { extends: "p" }
);
```

Remarque

Il est possible que vous rencontriez dans d'autres ressources une référence à `Document.registerElement()`. Cette méthode est obsolète, puisque faisant partie de l'API v0.

2.2 Éviter les conflits

La chaîne de caractères donnée en premier argument à `customElements.define()` définit donc le nom de l'élément. Plusieurs restrictions s'appliquent.

Avant tout, le nom utilisé se doit d'être unique. Si un autre élément personnalisé a été précédemment enregistré avec ce même nom, une `DOMException NotSupportedError` est levée. Il est donc impossible de "remplacer" un élément déjà défini.

Par souci de simplicité, aucun mécanisme spécifique (comme des espaces de noms XHTML) n'a été défini pour éviter ce problème.

Le standard exige cependant que tout nom d'élément personnalisé contienne un trait d'union (sans quoi une `SyntaxError` est levée). Cela afin notamment de garantir une compatibilité ascendante, étant donné qu'à l'avenir, aucun élément ne sera ajouté à HTML, à SVG ou à MathML avec un nom contenant des traits d'union.

Par convention, le préfixe du nom de l'élément personnalisé (soit `hello` dans nos exemples précédents) est considéré comme son "espace de noms".

La meilleure solution consiste donc à définir un nom court (sigle ou mot-clé) pour votre organisation, votre collection de Web Components ou votre projet, et à l'utiliser comme espace de noms. Bien entendu, il est préférable de vous assurer que ce préfixe n'est pas déjà exploité par une collection que les utilisateurs de votre élément personnalisé soient susceptibles d'utiliser. Pour cela, vous pouvez notamment explorer webcomponents.org et NPM.

Remarque

Les paquets NPM associés à des éléments personnalisés sont souvent étiquetés avec les mots-clés "web-components" et "custom-elements" (parfois sans trait d'union). Vous pouvez utiliser ces mots-clés pour faciliter votre recherche.

Voici quelques exemples de collections de Web Components communément utilisées :

Projet	Préfixe
vanillawc	`wc-`
Vaadin	`vaadin-`
Lion Web Components	`lion-`
Material Web Components	`mwc-`
@ionic/core	`ion-`
PolymerElements	`iron-`, `paper-`
Onsen UI	`ons-`

Dans certains cas, il peut être préférable de simplement exporter (sous forme d'ES module) la classe de définition de votre custom element, plutôt que de directement faire appel à `customElements.define()` dans le même fichier. Vous laissez ainsi aux développeurs tiers la liberté de définir le nom de leur choix en faisant eux-mêmes appel à `customElements.define()`. Compte tenu du potentiel risque de mauvaise configuration, il est cependant préférable d'éviter cette solution pour les éléments intégrés personnalisés.

2.3 Utiliser un nom valide

En plus de devoir contenir un trait d'union, le nom d'un élément personnalisé doit impérativement :

– commencer par une lettre minuscule non accentuée ([a-z]), sans quoi le parser HTML est susceptible de l'interpréter comme du texte et non comme un élément (tag) ;

– ne contenir aucune lettre majuscule ([A-Z]), les navigateurs devant être capables de traiter les éléments HTML sans sensibilité à la casse ;

– ne contenir aucun caractère non autorisé.

Cette dernière règle peut être déroutante, car de très nombreux caractères peuvent être utilisés en HTML.

Dans la très grande majorité des cas, il est donc préférable de se restreindre aux lettres minuscules, chiffres, trait d'union et underscore. Même si, en théorie, l'utilisation par exemple d'un point médian (·), de lettres grecques (telles que Ω), voire cyrilliques, demeure conforme au standard. Par ailleurs, la plupart des symboles (scientifiques, alphanumériques, etc.) sont exclus.

Cette règle précise a avant tout pour but d'assurer que l'élément personnalisé défini puisse toujours être créé par un appel à `createElement()` ou à `createElementNS()`.

Cependant, la grande majorité des navigateurs tendent à ne pas précisément respecter le standard quand les caractères utilisés sortent d'un cadre "classique". C'est par exemple le cas si vous tentez d'utiliser un émoji ou un autre caractère spécial théoriquement valide.

Ainsi, l'élément personnalisé autonome `<emotion-😍>` donné comme un exemple de nom valide par le standard peut généralement être défini et utilisé sans erreur. Cependant, une erreur est levée si vous tentez d'en créer une instance via un appel à `createElement('emotion-😍')`. Ce qui peut être généralement contourné en utilisant la propriété `innerHTML` comme dans l'exemple suivant.