

Développement et architecture des Applications **Web Modernes**

Retrouver les fondamentaux

Préface de **Hubert Sablonnière** - Développeur et conférencier spécialiste du Web
Postface de **Wassim Chegham** - Sr. JavaScript Developer Advocate Engineer
chez Microsoft

Noël MACE

En téléchargement



code source



compléments sur le site
fullweb.dev





Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence de l'ouvrage **DPAWM** dans la zone de recherche
et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Préface

Déconstruire pour ensuite mieux (re)construire

Introduction

- 1. Pourquoi ce livre ? 13
 - 1.1 État des lieux. 13
 - 1.2 Approche générale 14
 - 1.3 Modernité et minimalisme 15
 - 1.4 À qui s'adresse ce livre ? 16
- 2. Contenu 17
 - 2.1 Déroulement 17
 - 2.2 Chapitres. 17
 - 2.3 Conseils de lecture 21

Chapitre 1

Comprendre l'histoire du Web et de ses standards

- 1. Être un "bon citoyen" du Web 23
- 2. IETF, les réseaux et Internet 25
 - 2.1 Origines de la standardisation des réseaux 25
 - 2.2 Indépendance et internationalisation 26
 - 2.3 RFC : un fondamental de la collaboration 27
- 3. Invention de l'hypertexte. 30
 - 3.1 Modéliser la connaissance 30
 - 3.2 Première implémentation. 32

2 Applications Web Modernes

Retrouver les fondamentaux

4.	Origines du Web	35
5.	W3C, standardiser et promouvoir le Web	38
5.1	Origines (1991-1997)	38
5.2	Fonctionnement	40
5.3	Organisation générale.	42
5.4	Préévaluation par la communauté.	43
5.5	Élaboration des recommandations	47
5.6	Working groups	50
6.	WHATWG, HTML et DOM	51
6.1	Critiques et dissensions au W3C.	51
6.2	Le Web comme plateforme de développement applicatif	54
6.3	Ouverture et réconciliation	56
6.4	Domaines d'autorité.	57
6.5	Développement continu.	59
6.6	Processus et hiérarchie	61
7.	TC39, de JavaScript à ECMAScript.	62
7.1	Première standardisation	65
7.2	Évolutions de JavaScript	66
7.3	Moderniser le développement web pour de bon	69
7.4	Processus actuel	71
7.5	Organisation	73
8.	Références	75

Chapitre 2
Suivre les évolutions de la plateforme

- 1. Les précurseurs 85
 - 1.1 Les premiers navigateurs (1993-1994) 85
 - 1.2 Dynamiser le Web (1994-1995) 90
- 2. De Netscape à Internet Explorer 92
 - 2.1 La première guerre des navigateurs (1995-2001)..... 92
 - 2.2 Naissance de Mozilla (1998-2004)..... 96
 - 2.3 Internet Explorer contre Firefox (2004-2010)..... 100
- 3. Naissance de Chrome et de Safari 102
- 4. L'entrée dans le Web moderne 105
 - 4.1 Le succès des smartphones (2011-2017) 105
 - 4.2 Le règne de Google (2017-?) 107
 - 4.3 Situation actuelle (2021) 111
- 5. À l'écoute des nouveaux standards 112
 - 5.1 L'inintelligibilité supposée des standards 112
 - 5.2 Trouver et suivre les propositions 116
- 6. Progresser avec les navigateurs modernes 120
 - 6.1 Trois approches distinctes 120
 - 6.2 Google Chrome 121
 - 6.3 Apple Safari et WebKit 122
 - 6.4 Mozilla et Firefox 123
 - 6.5 Veille et montée en compétences 126
 - 6.6 Perspectives présentes et futures 128
- 7. Références 130

4 Applications Web Modernes

Retrouver les fondamentaux

Chapitre 3

Adopter une approche par composant

1. Introduction à la programmation par composant	137
1.1 Un cadre de référence nécessaire	138
1.2 Niveau de complexité et réutilisabilité	139
1.3 Qu'est-ce qu'un composant ?	140
1.4 Définir un composant	143
1.5 Les limites initiales du développement web	145
2. Premières fondations pour le Web	148
2.1 Les plug-ins	148
2.2 Une seconde génération de frameworks	152
3. Cas d'étude : AngularJS	154
4. Liens avec une approche fonctionnelle	158
4.1 Fonctions et réactivité	159
4.2 Approche générale	160
4.3 Représenter HTML dans JavaScript	162
4.4 Problème du rendering	166
4.5 Problème de la gestion d'état	169
4.6 Vers un langage fonctionnel ?	174
5. Naissance d'un standard	177
5.1 Les prédécesseurs	178
5.2 Première émergence des Web Components	180
5.3 Introduction aux Web Components	181
5.4 Anticiper et dépasser le standard	183
6. Références	187

Chapitre 4
Faire bon usage des Web Components

- 1. Introduction 195
- 2. Étendre le vocabulaire HTML 197
 - 2.1 Définir un élément personnalisé 197
 - 2.2 Éviter les conflits 199
 - 2.3 Utiliser un nom valide 201
 - 2.4 Créer une classe 204
 - 2.5 Afficher un message 206
 - 2.6 Créer un élément personnalisé programmatiquement 206
 - 2.7 Cycle de vie 210
 - 2.8 Support 214
- 3. Créer et utiliser un arbre fantôme 217
 - 3.1 Notions fondamentales 217
 - 3.2 light DOM et shadow DOM 221
 - 3.3 Restreindre l'accès au shadow DOM 224
- 4. Définir le style d'un Web Component 228
 - 4.1 Style externe 230
 - 4.2 Élément hôte 238
 - 4.3 Style par défaut de l'élément hôte 240
 - 4.4 Contexte 242
 - 4.5 Customisation 244
- 5. Templates et composition 246
 - 5.1 <template> 247
 - 5.2 <slot> et composition 252
 - 5.3 Slots nommés 255

6 Applications Web Modernes

Retrouver les fondamentaux

6.	Attributs et propriétés	258
6.1	Principe : suivre la logique du standard HTML	258
6.2	Mise en place d'un Web Component de démonstration	259
6.3	Exposer l'état du composant	261
6.4	Effectuer des actions simples	262
6.5	Récupérer des valeurs primitives	264
6.6	Réagir aux changements de valeur d'un attribut	266
6.7	Manipuler des valeurs non primitives	268
6.8	Charger des propriétés dynamiquement	270
7.	Limites des Web Components	272
8.	Références	275

Chapitre 5

Développer une application monopage

1.	Structure et objectifs	279
1.1	Les défauts des sites web multipages	279
1.2	Offrir une meilleure expérience utilisateur	281
1.3	Architecture	283
2.	Principes essentiels du routage	287
3.	Manipuler l'URL du document	289
3.1	L'interface Location	291
3.2	Utiliser l'identificateur de fragment de l'URL	293
3.3	Hash et historique	294
3.4	L'interface History	296
3.5	Compatibilité	301
3.6	Réécriture des URL côté serveur	303
4.	Définir des routes	306
4.1	Uniformiser le rendu dynamique	307
4.2	Définir une collection de routes	309
4.3	Définir une route par défaut	311
4.4	Approche des principaux frameworks	315

- 5. Transmettre des données 317
 - 5.1 Paramètres de route 318
 - 5.2 Paramètres d'URL 323
 - 5.3 Objets état associés à l'historique 328
- 6. Navigation et changement de route 331
 - 6.1 Gestion globale 333
 - 6.2 Évènement popstate 336
- 7. Au-delà du routage 338
 - 7.1 Modularité 338
 - 7.2 Limitations du routage 341
- 8. Références 342

Chapitre 6
Afficher des données dynamiquement

- 1. Optimiser le chargement initial 347
 - 1.1 Faire primer l'expérience utilisateur 347
 - 1.2 Capturer l'attention des utilisateurs 348
 - 1.3 Audit des performances du chargement initial 350
 - 1.4 Éviter les contenus bloquants 355
 - 1.5 Améliorer le chargement des scripts 356
 - 1.6 Accélérer l'affichage d'un texte 357
 - 1.7 Ne charger que les styles utiles 358
 - 1.8 Au-delà du simple chargement 360
- 2. Comparer différentes approches 361
 - 2.1 Présentation de la solution employée 367
 - 2.2 Implémentation des tests de performance 369
 - 2.3 Est-il vraiment important de comparer les performances ? . . . 374
- 3. Bien utiliser la DOM API 376
 - 3.1 Créer un ensemble d'éléments 376
 - 3.2 Mettre à jour le contenu d'un élément 378

8 Applications Web Modernes

Retrouver les fondamentaux

4.	Littéraux de gabarit et innerHTML.	382
4.1	Gérer les évènements	383
4.2	Mettre à jour le contenu de l'élément.	388
4.3	Performances.	390
5.	HyperScript.	393
5.1	Une bibliothèque, une fonction et une syntaxe.	394
5.2	HyperScript, React et les VDOM	396
5.3	Syntaxes alternatives pour HyperScript	399
5.4	Créer une fonction HyperScript	400
5.5	Performances.	402
6.	Minimiser les accès au DOM.	404
6.1	Identifier les responsabilités de chacun.	405
6.2	Fragments	407
6.3	Mettre en cache les éléments.	410
6.4	Liens avec la programmation orientée composant.	415
6.5	Comparaison avec les littéraux de gabarits.	419
6.6	Mettre en cache un élément avec HyperScript.	421
6.7	Mettre en cache l'état.	425
7.	Optimiser pour la répétition	426
7.1	Optimiser la mise en cache des éléments avec cloneNode()	427
7.2	Utiliser innerHTML pour améliorer la lisibilité	431
7.3	Construire sa propre solution	433
7.4	Performances.	442
8.	Conclusion	443
9.	Références	444

Chapitre 7

Définir une architecture cohérente

- 1. Comment faire "le bon choix" ? 449
 - 1.1 Déterminer l'approche optimale en fonction du contexte. 450
 - 1.2 Trouver le bon équilibre entre abstraction et dette technique. 452
- 2. Vanilla Web et les microbibliothèques 455
 - 2.1 De Vanilla JS au modern Web 455
 - 2.2 Complexité et poids d'un code JavaScript 456
 - 2.3 Évolution d'une bibliothèque dans le temps. 458
 - 2.4 Quand et pourquoi utiliser des microbibliothèques. 459
 - 2.5 Aides à la manipulation du DOM 460
 - 2.6 Réalité du "Vanilla". 464
 - 2.7 RE:DOM 466
- 3. Bibliothèques de rendering. 476
 - 3.1 Principe des étiquettes de gabarits. 477
 - 3.2 Bibliothèques de rendering utilisant les étiquettes de gabarits 480
 - 3.3 Naissance et évolution de lit-html 484
 - 3.4 Fonctionnement de lit-html 485
 - 3.5 Exemple d'application 487
 - 3.6 Les directives lit-html. 492
 - 3.7 JSX face aux littéraux de gabarits 494
 - 3.8 React est-il vraiment une bibliothèque de rendering ? 502
- 4. Modularité et programmation orientée composant. 504
 - 4.1 Principes de lit-element 506
 - 4.2 Démarrer un projet avec lit-element 510
 - 4.3 web component par compilation avec Stencil 514
- 5. Programmation réactive 516
 - 5.1 Notions fondamentales 516
 - 5.2 RxJS 519
 - 5.3 Observables et opérateurs 520

10 Applications Web Modernes

Retrouver les fondamentaux

5.4	Combiner des observables	524
5.5	hot et cold observables	525
5.6	Alternatives à RxJS	527
5.7	Forces et faiblesses de la programmation réactive	528
6.	Gestion d'état	529
6.1	Créer un conteneur d'état avec Redux	530
6.2	Permettre à des composants de partager un état	533
6.3	Rendre les composants indépendants de l'état global	536
7.	Routeurs	541
7.1	Page.js	541
7.2	Universal-router	543
8.	Frameworks	545
8.1	Définition	545
8.2	Microframeworks	547
8.3	React, Vue et Angular : frameworks ou écosystèmes ?	548
8.4	Metaframeworks	552
9.	Synthèse	553
9.1	Vanilla JS et Vanilla Web : comment développer sans assistance ?	555
9.2	Le modern Web face aux compilateurs et aux frameworks.	556
10.	Références	557
Conclusion		569
Postface		575

Annexes

1. Les bibliothèques, les frameworks et les outils	583
1.1 Le rendering	584
1.1.1 Les manipulations du DOM	584
1.1.2 Les tagged template literals	585
1.1.3 JSX et React	585
1.2 La programmation orientée composant	586
1.3 Le routage et la navigation	586
1.4 La programmation réactive et les gestionnaires d'état	587
1.5 Les frameworks	587
1.6 Les outils de développement	587
1.7 Les polyfills et les shims	588
2. Groupes de travail du W3C	589
2.1 Note à propos des abréviations	589
2.2 Fondamentaux front-end	590
2.3 Accessibilité (WAI)	591
2.4 Web API	592
2.5 Sémantique et données	593
2.6 Divers	593
2.7 Références	595
3. Glossaire	597
4. Principes fondamentaux de Chrome	603
4.1 Multiprocessing	603
4.2 V8, l'interpréteur JavaScript	604
4.3 Expérience utilisateur	605
4.4 Sécurité	605
4.5 Gears, standards et open source	606

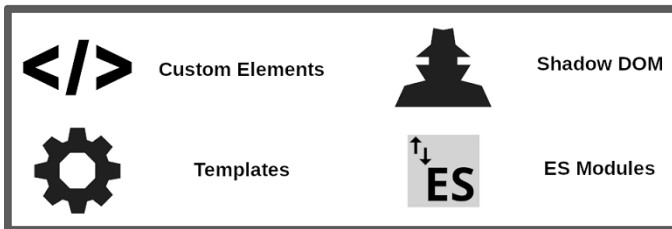
Chapitre 4

Faire bon usage des Web Components

1. Introduction



Web Components



HTML Imports

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 besoins. 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.whenDefined()`, 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 pourrions 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](https://www.npmjs.com/).

■ 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 emoji 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.