



# Beste de savoir

Apprendre le JavaScript moderne en  
créant une to-do list de A à Z

---

26 avril 2022



# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1. La syntaxe</b>	<b>4</b>
Introduction	4
1.1. Les variables et constantes	4
1.1.1. <code>var</code> pour les variables à grande portée	4
1.1.2. <code>let</code> pour les variables locales	5
1.1.3. <code>const</code> pour les valeurs fixes	5
1.2. Les littéraux de gabarits	6
1.2.1. Les sauts de ligne	6
1.2.2. L'interpolation de données	6
1.2.3. Les <i>étiquettes</i> ou <i>tags</i>	8
1.3. Les fonctions fléchées	8
1.4. Les modules	9
1.5. Les class	10
Conclusion	12
<b>2. Les Web Components</b>	<b>13</b>
Introduction	13
2.1. Créer et déclarer un composant	13
2.1.1. Hériter d'un élément existant	13
2.2. Utiliser un composant	14
2.2.1. Avec du HTML	14
2.2.2. Avec du JavaScript	14
2.3. Récupérer les attributs	14
Conclusion	15
<b>3. Les événements</b>	<b>16</b>
Introduction	16
3.1. Les événement natifs	16
3.2. Les custom events	17
<b>4. Construire la todo-list</b>	<b>18</b>
Introduction	18
4.1. Un composant dynamique commun	18
4.2. Une étiquette pour les templates	19
4.3. Nos premiers composants interactifs	20
4.3.1. La racine de l'application	21
4.3.2. La création de liste	22
4.4. Remonter les informations	25
Conclusion	25

<b>5. Le stockage</b>	<b>26</b>
Introduction . . . . .	26
5.1. Le stockage local (localStorage) . . . . .	26
5.2. Communiquer avec un serveur . . . . .	26
Conclusion . . . . .	26
<b>Conclusion</b>	<b>27</b>

# Introduction

*Vous êtes perdu parmi les frameworks front à la mode mais vous voulez garder votre code léger et éviter de dépendre d'un tas de fonctions obscures ?*

*Vous avez créé vos premières pages Web statiques et vous voudriez les rendre plus dynamiques, pour interagir avec l'utilisateur pour facilement ?*

**Pas de panique, vous êtes au bon endroit !**

Nous allons donc apprendre ensemble à créer une *application Web* (aussi appelée *web app*) en **JavaScript nouvelle génération** (en l'occurrence ES10 ou ES2019) pour créer notre propre to-do list !



## Pré-requis

Ce tutoriel a pour objectif de vous faire découvrir le JavaScript moderne. Vous aurez besoin de quelques notions de HTML et CSS (de quoi créer des pages Web basiques, en gros) et de développement (idéalement en JavaScript, mais n'importe quel langage devrait faire l'affaire), le reste sera du bonus.



Si vous ne comprenez pas une notion n'hésitez pas à faire quelques recherches ou à demander une clarification sur le forum. C'est l'avantage du Web : toute la documentation et les ressources pour apprendre sont disponibles via votre moteur de recherche préféré, sinon il y a aussi des gens sympas pour vous guider !

# 1. La syntaxe

## Introduction

Avant de s'attaquer au cœur du sujet, il est primordial de voir quelques bases sur la syntaxe JavaScript moderne.

Voyons donc ensemble les nouveautés en JS, et ce qu'elles peuvent nous apporter pour un code plus simple et moderne.

*i*

Si vous avez un doute sur le support d'une fonctionnalité (puisque'il s'agit ici de choses assez récentes), le site [Can I use](#) permet de voir rapidement les navigateurs qui la supportent ou non.

### 1.1. Les variables et constantes

Si vous avez déjà fait un peu de développement, vous êtes probablement déjà familiers avec le concept des variables.

Pour faire simple, ce sont des noms que vous utilisez pour stocker des données (qu'il s'agisse de *chaînes de caractère*, de *nombres*, de *booléens* ou même *d'objets complexes*).

#### 1.1.1. `var` pour les variables à grande portée

Historiquement, en JavaScript on utilisait le mot-clé `var` pour déclarer une variable :

```
1 var bonjour // Déclaration de la variable
2 bonjour = 'Bonjour tout le monde' // Initialisation de la variable
3
4 var age = 42 // Déclaration et initialisation en une seule
   instruction
```

Le problème de ces variables est leur portée : elles sont valides pour l'ensemble de la fonction qui les contient. Il devient donc difficile de les déclarer dans un bloc qui n'est pas une fonction, comme une boucle :

## 1. La syntaxe

```
1 for (var i=1; i<=10; i++) {
2     var j = i // 1, 2, 3 ... 10
3 }
4 j // 10
5 i // 11
```

On le voit ici : la variable `j` a été déclarée dans la boucle, mais est accessible à l'extérieur et risque par exemple de rentrer en conflit avec une autre variable `j`.

Pour cette raison, il existe maintenant un mot-clé plus pratique qui remplace peu à peu `var`...

### 1.1.2. `let` pour les variables locales

Tout comme `var`, le mot-clé `let` permet de déclarer une variable. La différence est sa portée, qui est limitée au bloc qui la contient :

```
1 for (let i=1; i<=10; i++) {
2     let j = i // 1, 2, 3 ... 10
3 }
4 j // undefined
5 i // undefined
```

Les variables ne sont ainsi accessibles *que* dans le bloc (ici délimité par des accolades) qui les contient (ainsi que leurs enfants), sans écraser des variables d'un bloc supérieur.

### 1.1.3. `const` pour les valeurs fixes

Et si vous voulez enregistrer une valeur qui ne changera jamais ? Vous savez, une *constante* ! 🍊

Et bien, il existe le mot-clé `const` qui fonctionne à peu près comme `let`, sauf qu'il ne vous laisse pas modifier la valeur vers laquelle le nom pointe.

```
1 const AGE_MINIMUM = 7
2 const AGE_MAXIMUM = 77
3
4 AGE_MINIMUM-- // Invalide, la valeur ne peut pas être remplacée
5 AGE_MAXIMUM = 100 // Invalide, la valeur ne peut pas être remplacée
```



Attention : vous ne pouvez pas modifier le *pointeur* (la référence mémoire), mais vous pouvez modifier l'objet contenu dans votre constante :

## 1. La syntaxe



```
1 const ANIMAUX = [' ', ' ', ' ']  
2 ANIMAUX.push(' ', ' ') // 5  
3 // Le tableau contient maintenant 5 éléments sans erreur, car  
   // il s'agit d'un objet et non d'une valeur primaire
```

Si vous voulez en apprendre plus sur les déclarations en JS, n'hésitez pas à lire [la page MDN sur le sujet](#) .

## 1.2. Les littéraux de gabarits

Également appelés *templates literals* pour les intimes, il s'agit d'une nouvelle façon de créer des chaînes de caractère, avec plusieurs avantages à la clé...

### 1.2.1. Les sauts de ligne

À la différence des chaînes classiques, délimitées par des *quotes* (simples ou doubles), vous pouvez sauter une ligne sans problème dans un littéral de gabarit :

```
1 let singleQuotes = '  
2 ' // SyntaxError: ' ' string literal contains an unescaped line  
   // break  
3  
4 let doubleQuotes = "  
5 " // SyntaxError: "" string literal contains an unescaped line  
   // break  
6  
7 let templateLiteral = `  
8 ` // OK
```

### 1.2.2. L'interpolation de données

Fini les concaténations sans fin ! Si vous voulez injecter une variable (ou une constante) dans un littéral de gabarit, il existe une syntaxe spécifique pour ne pas avoir à concaténer, en utilisant le marqueur `${}` autour de votre valeur :

```
1 const NUMERO = '007'  
2
```

## 1. La syntaxe

```
3 let message = `Bonjour, Agent ${NUMERO}` // "Bonjour, Agent 007"
```

Et ça fonctionne aussi avec une expression, un appel de fonction ou une propriété d'objet :

```
1 const auteur = {
2     pseudo: 'viki53',
3     espece: '',
4     tutoriels: 4
5 }
6
7 function nomEspece(code) {
8     switch (code) {
9         case '':
10            return 'un dauphin'
11        case '':
12            return 'un loup'
13        case '':
14            return 'un caribou'
15        case '':
16            return 'un renard'
17        default:
18            return 'un animal inconnu'
19    }
20 }
21
22 let presentation = `Ceci est le ${auteur.tutoriels +
1}e tutorial de ${auteur.pseudo}, qui est ${nomEspece(auteur.espece)}` // "Ceci est le 5e tutorial de viki53, qui est un dauphin"
```



Les valeurs sont évaluées au moment de la déclaration, pas à chaque fois que vous faites appel à votre chaîne. Si vous changez une valeur après avoir déclaré la chaîne il faudra la redéfinir.

```
1 let langage = 'JavaScript'
2
3 let message = `Le meilleur langage pour le Web est ${langage}`
4
5 langage = 'Python'
6
7 message // "Le meilleur langage pour le Web est le JavaScript"
```

## 1. La syntaxe

### 1.2.3. Les étiquettes ou tags

En plus de pouvoir interpoler des variables, vous pouvez aussi récupérer chaque morceau des chaînes pour modifier la chaîne finale via une étiquette :

```
1 const nom = 'viki53'  
2 const metier = 'développeur'
```

Ces étiquettes sont des fonctions, qui doivent répondre à une définition précise : le premier paramètre est un `Array` contenant les morceaux bruts de la chaîne (en dehors des marqueurs d'interpolation), les autres paramètres contenant les valeurs à injecter :

```
1 function majuscules(morceaux, ...valeurs) {  
2     let str = ''  
3  
4     for (let i in morceaux) {  
5         str += morceaux[i] + (valeurs[i] ||  
6             '').toUpperCase()  
7     }  
8     return str  
9 }
```

On peut ensuite appliquer le *tag* à notre *template string* :

```
1 const direBonjour =  
    majuscules`Bonjour, je m'appelle ${nom} et je suis ${metier}`  
    // "Bonjour, je m'appelle VIKI53 et je suis DÉVELOPPEUR"
```

### 1.3. Les fonctions fléchées

Les *arrow functions* sont des fonctions comme les autres, à un détail près : elles n'ont pas de `this` spécifique. Elles permettent également une syntaxe plus courte, pratique pour déclarer des fonctions anonymes.

```
1 function classique() {  
2     console.dir(this) // body  
3 }  
4 document.body.addEventListener('click', classique)  
5
```

## 1. La syntaxe

```
6 const flechee = () => {  
7     console.dir(this) // Window  
8 }  
9 document.body.addEventListener('click', flechee)
```

*i*

Si vous avez un doute sur l'usage de `this` ou ce qu'il définit, vous pouvez trouver une explication sur le [MDN](#) ↗.

Ça peut paraître abstrait pour le moment, mais vous verrez en construisant l'application que c'est très utile 🍊

À noter que les fonctions (fléchées ou non) n'ont pas forcément besoin d'un nom, elles peuvent être *anonymes* :

```
1 document.documentElement.addEventListener('scroll', event => {  
    console.dir(event) }, { once: true, passive: true })
```

### 1.4. Les modules

Une des nouveautés du JavaScript moderne est l'apparition des modules, qui permettent de structurer son code en séparant chaque fonctionnalité ou composant indépendamment du reste : chaque module gère son aspect métier sans se soucier des autres.

Chaque module est exécuté dans un contexte différent: vous ne risquez donc pas d'avoir des conflits de variables entre deux fichiers. Un module peut donc, en règle générale, être transposé d'un projet à l'autre directement sans risque.

Avant de pouvoir charger un module, il faut d'abord en exporter un pour le rendre accessible aux autres:

```
1 // Cette fonction sera chargée par défaut si l'import ne précise  
    pas d'importer autre chose  
2 export default function maFonction() {  
3     console.info('Bonjour !')  
4 }
```

Listing 1 – ma-fonction.js

L'export permet d'exposer une valeur afin que les autres modules puissent y accéder. Sans export la valeur reste interne au module, les autres ne peuvent donc pas y accéder directement.

Vous pouvez également exporter un ensemble de variables et fonctions :

## 1. La syntaxe

```
1 export const auteur = {
2     pseudo: 'viki53',
3     espece: ''
4 }
5
6 export const meilleureEspece = ''
7
8 export function nomEspece(code) {
9     switch (code) {
10        case '':
11            return 'un dauphin'
12        case '':
13            return 'un loup'
14        case '':
15            return 'un caribou'
16        case '':
17            return 'un renard'
18        default:
19            return 'un animal inconnu'
20    }
21 }
```

Listing 2 – un-module.js

Pour utiliser un module, il faut déclarer au navigateur que l'on veut charger notre fichier dans ce contexte, via un attribut HTML :

```
1 <script src="./js/mon-app.js" type="module"></script>
```

Notre JavaScript pourra alors à son tour charger d'autres modules, via une syntaxe spécifique :

```
1 import maFonction from './ma-fonction.js' // Charge l'export par
   défaut (`default`) du module sous le nom `maFonction`
2 import * as monModule from './un-module.js' // Charge l'ensemble
   du module
3 import { auteur, nomEspece as emojiVersNomEspece } from
   './un-module.js' // Importe une partie du module, renomme un
   des imports
```



À noter que les chemins pour les imports sont calculés à partir du dossier courant.

### 1.5. Les class

Pour se rapprocher d'autres langages, le JS moderne a introduit les `class` pour définir des objets, avec des propriétés et des méthodes selon une syntaxe classique :

## 1. La syntaxe

```
1 class Personne {
2     constructor(nom) {
3         this.nom = nom
4     }
5
6     direBonjour() {
7         return `Bonjour, je m'appelle ${this.nom}`
8     }
9 }
```



### Attention

Il ne s'agit pas réellement d'objets, mais de prototypes (la syntaxe est convertie en prototype par le moteur JavaScript), comme dans les versions précédentes de JavaScript. Il s'agit surtout d'une couche syntaxique pour clarifier/simplifier le code.

```
1 const auteur = new Personne('viki53')
2 auteur.direBonjour() // "Bonjour, je m'appelle viki53"
```

Vous pouvez aussi utiliser l'héritage pour étendre une `class` parente (et une seule, pour le moment) :

```
1 class EtreVivant {
2     constructor() {}
3
4     get espece() {
5         return this._espece
6     }
7 }
8
9 class Dauphin extends EtreVivant {
10     constructor() {
11         super()
12         this._espece = ''
13     }
14 }
15
16 const viki53 = new Dauphin()
17 viki53.espece // ""
```

## 1. La syntaxe

### Conclusion

*i*

N'oubliez pas de vérifier le support des fonctionnalités que vous utilisez, par exemple via [Can I use](#) .

Par exemple pour les *template literals* ``` , IE11 ne les gère pas. Edge 13, ainsi que Safari 9.1 et supérieurs, les gèrent (avec un bug connu sur Safari 12).

Maintenant que vous savez (presque) tout sur la syntaxe, on va pouvoir passer aux choses sérieuses ! 🍌

## 2. Les Web Components

### Introduction

Saviez-vous que vous pouvez créer vos propres balises HTML ? Vous n'êtes pas limités à celles disponibles dans la norme HTML5, vous pouvez créer vos propres composants avec un peu de JavaScript !

On utilise pour cela des [Web Components](#) , qui vont utiliser le Shadow DOM pour vivre indépendamment de la page : ils ont en quelque sorte leur propre DOM dédié, tout en pouvant interagir avec l'extérieur si besoin.

### 2.1. Créer et déclarer un composant

Pour créer un composant il faut indiquer au navigateur son nom (qui servira par exemple dans notre HTML), mais aussi la `class` JavaScript à initialiser pour chaque instance de notre composant :

```
1 window.customElements.define('my-app', AppElement)
```

À quoi ressemble cette fameuse `class` ? Et bien de base elle est plutôt simple, puisqu'elle doit juste hériter de `HTMLElement` :

```
1 export class AppElement extends HTMLElement {  
2 }
```

Ne vous inquiétez pas, elle ne restera pas aussi vide bien longtemps... On va voir tout de suite comment la rendre plus dynamique et utile !

#### 2.1.1. Hériter d'un élément existant

Un composant peut aussi étendre les capacités d'une balise HTML ou d'un composant existant :

```
1 export class CustomParagraphElement extends HTMLParagraphElement {  
2 }  
3 window.customElements.define('my-p', CustomParagraphElement, {  
   extends : 'p' })
```

### 2.2. Utiliser un composant

Pour utiliser notre composant, il y a deux méthodes principales :

#### 2.2.1. Avec du HTML

Votre composant étant déclaré par votre JavaScript, vous pouvez utiliser son nom comme une balise HTML classique :

```
1 <my-app></my-app>
```

Sans oublier de charger le JS qui correspond :

```
1 <script type="module" src="app.element.js"></script>
```

##### 2.2.1.1. Les héritiers d'éléments existants

Si votre composant étend un élément existant vous pouvez utiliser la balise de base avec l'attribut `is` pour préciser le nom de votre composant :

```
1 <p is="my-p"></p>
```

#### 2.2.2. Avec du JavaScript

Vous pouvez insérer votre composant comme un élément du DOM classique, en ajoutant une instance dans un nœud parent :

```
1 import { AppElement } from 'app.element.js' // On importe notre
   élément
2
3 const composant = new AppElement() // On crée une instance de
   l'élément
4
5 document.body.appendChild(composant) // On injecte l'élément dans
   le DOM
```

### 2.3. Récupérer les attributs

?

C'est bien beau tout ça, mais comment j'injecte des valeurs à mon composant, moi ?!

Eh bien en définissant des attributs, pardi !

## 2. Les Web Components

On va d'abord déclarer quels attributs on veut surveiller (histoire que le navigateur nous prévienne quand ils changent) grâce à une propriété statique (commune à toutes les instances, donc) `observedAttributes` sur notre composant :

```
1 static get observedAttributes() {  
2     return ['titre'] // On va pouvoir ajouter d'autres  
    attributs à cette liste si besoin  
3 }
```

En cas de changement, le navigateur va alors appeler la méthode `attributeChangedCallback` de notre composant en précisant le nom de l'attribut, la valeur d'origine et la nouvelle valeur :

```
1 attributeChangedCallback(nom, ancienneValeur, nouvelleValeur) {  
2     if (nom === 'titre') {  
3         this.titre = nouvelleValeur  
4     }  
5 }
```

## Conclusion

Maintenant que vous savez créer vos propres composants, il est temps de les faire communiquer pour obtenir une *todo-list* !

## 3. Les événements

### Introduction

Il existe, en JavaScript, des événements natifs qui permettent de détecter des actions sur les éléments du DOM comme le clic, le mouvement de la souris ou le *scroll*... mais saviez-vous que vous pouvez aussi définir vos propres événements ?

### 3.1. Les événement natifs

Revoyons d'abord comment capturer un événement :

```
1 const lien = document.querySelector('#titre > a')
2
3 lien.addEventListener('click', (event) => {
4     event.preventDefault()
5 })
```

Plutôt simple, n'est-ce pas ?

Et si on voulait récupérer l'événement ailleurs, tout en s'assurant que ce même lien a été cliqué ?

```
1 const lien = document.querySelector('#titre > a')
2
3 document.addEventListener('click', (event) => {
4     if (lien === event.originalTarget) {
5         // Le lien a été cliqué !
6     }
7 })
```



Attention, si votre cible contient des éléments enfants, ce seront peut-être eux la cible d'origine de l'événement !

```
1 document.addEventListener('click', (event) => {
2     if (lien === event.originalTarget ||
3         lien.contains(event.originalTarget)) {
4         // Le lien ou un de ses descendants a été cliqué !
5     }
6 })
```

## 3.2. Les custom events

Si vous souhaitez diffuser des événements personnalisés, pour diffuser des informations propres à vos composants, il existe une `class` nommée `CustomEvent` :

```
1 const monEvenement = new CustomEvent('bonjour')
2
3 this.dispatchEvent(monEvenement) // On diffuse l'événement
```

Si vous avez besoin de passer des informations plus détaillées, vous pouvez renseigner l'option `detail` :

```
1 const monEvenement = new CustomEvent('bonjour', {
2     bubbles: true, // Indique à l'événement qu'il doit
3     detail: '' // On peut passer n'importe quelle valeur dans
4     cette propriété
5 })
```

Et les récupérer via la même propriété de l'événement :

```
1 document.addEventListener('bonjour', (event) => {
2     event.detail // ""
3 })
```

Si votre événement doit traverser le *Shadow DOM* il faudra penser à utiliser l'option `composed`, sinon il sera uniquement diffusé au sein de votre composant :

```
1 const monEvenement = new CustomEvent('bonjour', {
2     composed: true,
3     detail: ''
4 })
```

## 4. Construire la todo-list

### Introduction

Maintenant que vous maîtrisez le JavaScript moderne, voyons comment construire une application *front* rapidement !

#### 4.1. Un composant dynamique commun

Avant de construire nos composants personnalisés, il est important de formaliser leur fonctionnement.

On va donc créer une `class` commune pour structurer tout cela et faciliter le développement futur :

```
1 export class MyBaseElement extends HTMLElement {
2   constructor() {
3     super();
4     this.attachShadow({ mode: 'closed' }) // Notre
      Shadow DOM n'a pas besoin d'être modifié de
      l'extérieur
5     this.render()
6   }
7
8   connectedCallback() {
9     this.update() // Le composant est ajouté au DOM,
      on affiche le contenu
10  }
11
12  attributeChangedCallback(name, oldValue, newValue) {
13    this[name] = JSON.parse(unescape(newValue)) // On
      considère tous nos attributs comme du JSON
14    this.update() // On met à jour le contenu du
      composant pour ré-évaluer le template
15  }
16
17  update() {
18    this.shadowRoot.innerHTML = this.render() // On
      évalue le template
19  }
20
21  render() {
22    return '' // Notre composant est vide par défaut
23  }
}
```

#### 4. Construire la todo-list

```
24 }
```

Listing 3 – tools/my-base-element.js

Nos composants héritant de cette `class` seront donc automatiquement mis à jour en cas de changement de valeur d'un attribut ! 🍊

Et puisqu'on y est, gérons aussi les styles !

```
1 export class MyBaseElement extends HTMLElement {
2   get styles() { // Les styles ne devraient pas être trop
3     // dynamiques, on peut donc définir un simple _getter_
4     return '' // Chaque composant pourra définir ses
5     // propres styles
6   }
7
8   // [...] le constructeur, des callbacks...
9
10  updateStyles() {
11    const style = document.createElement('style')
12    style.textContent = this.styles
13    this.shadowRoot.appendChild(style)
14  }
15
16  update() {
17    // [...] le contenu actuel
18    this.updateStyles() // On ajoute cette ligne
19  }
20
21  // [...] le reste de la class
22 }
```

Listing 4 – tools/my-base-element.js

*i*

Si vous avez besoin d'un composant plus puissant, je vous recommande [LitElement](#) qui mettra à jour les valeurs de façon plus efficace et permet d'utiliser des templates plus complets.

## 4.2. Une étiquette pour les templates

Comme on utilisera des *template literals* dans nos composants, il peut être utile de faciliter l'injection de valeurs.

On a d'ailleurs préparé le terrain en évaluant le JSON dans les attributs dans la méthode `attributeChangedCallback` plus haut.

Automatisons donc tout cela dans nos templates avec un *tag* dédié :

#### 4. Construire la *todo-list*

```
1 export function html(strings, ...values) {
2   const l = strings.length - 1
3   let html = ''
4
5   for (let i = 0; i < l; i++) {
6     const s = strings[i]
7     let v = values[i]
8
9     if (Array.isArray(v)) {
10      v = v.join('')
11    }
12    if (typeof v === 'object') {
13      v = escape(JSON.stringify(v)) // La valeur
14      // à injecter est un objet, on la
15      // convertit alors en JSON et on
16      // l'échappe pour l'utiliser dans du HTML
17    }
18    html += s + v // On ajoute notre valeur au morceau
19    // correspondant
20  }
21  html += strings[l] // On n'oublie pas le dernier morceau
22
23  return html // On retourne la chaîne complète avec les
24  // valeurs injectées
25 }
```

Listing 5 – tools/custom-html.js

On pourra alors injecter des objets directement dans un *template* :

```
1 const valeur = 42
2 const objet = { foo: 'bar' }
3
4 const template = html`<mon-composant valeur="${valeur}" attribut="
5   ${objet}"></mon-composant>`
```

*i*

Si vous avez besoin d'un système de template plus complet (ou que vous utilisez `LitElement`) je vous recommande de jeter un œil à `lit-html` [↗](#), qui fournit une fonction du même style bien plus puissante.

### 4.3. Nos premiers composants interactifs

C'est parti pour commencer à avoir une application digne de ce nom !

## 4. Construire la todo-list

### 4.3.1. La racine de l'application

C'est le composant de base, celui qui va s'occuper de charger le plus gros de l'application et d'en gérer le fonctionnement global.

On va commencer léger, en affichant nos listes (qui sont pour l'instant au très grand nombre de... zéro ) et un formulaire pour en créer une (via un composant dédié).

```
1 import { MyBaseElement } from '../tools/my-base-element.js'
2 import { html } from '../tools/custom-html.js'
3 import { STYLE_COMMON } from '../tools/styles.js'
4
5 // On importe les composants utilisés dans le template
6 import { NewTodoListElement } from './new-todolist.element.js'
7 import { TodoListElement } from './todolist.element.js'
8
9 export class AppElement extends MyBaseElement {
10   get styles() {
11     return `
12       ${STYLE_COMMON}
13       :host {
14         display: flex;
15         flex-wrap: wrap;
16         width: 100%;
17         justify-content: center;
18       }
19
20       .todolist {
21         flex: 0 0 18rem;
22         max-width: calc(100% - 2rem);
23         margin: 1rem;
24         height: 25rem;
25         background-color: white;
26         color: black;
27
28         box-shadow: 0 3px 5px rgba(0, 0, 0, .3);
29         border-radius: .5rem;
30         overflow: hidden;
31       }
32     `;
33   }
34   constructor() {
35     super();
36
37     this.listes = [];
38   }
39
40   render() {
41     return `
42       ${this.listes && this.listes.length ?
```

#### 4. Construire la todo-list

```
43         this.listes.map(liste => html `<section is="todo-list" class="todolist" liste="${liste}"></section>`)
44         :
45     }
46 }
47 `
48     <section is="new-todo-list" class="todolist"></section>`
49 }
50 window.customElements.define('my-app', AppElement)
```

Listing 6 – elements/app.element.js



Notez l'utilisation du pseudo-sélecteur `:host` pour styliser notre composant, au lieu d'utiliser le nom de la balise : le CSS étant lui-aussi séparé de celui de la page.

#### 4.3.2. La création de liste

```
1 import { MyBaseElement } from '../tools/my-base-element.js'
2 import { html } from '../tools/custom-html.js'
3 import { STYLE_COMMON, STYLE_FORM } from '../tools/styles.js'
4
5 import { TodoList } from '../models/todolist.js'
6
7 /**
8  * @event newListName – Le nom de la liste à créer
9  */
10 export class NewTodoListElement extends MyBaseElement {
11     get styles() {
12         return `
13             ${STYLE_COMMON}
14             ${STYLE_FORM}
15             :host {
16                 display: flex;
17                 flex-direction: column;
18                 justify-self: stretch;
19                 align-self: stretch;
20             }
21             h1 {
22                 margin: 0;
23                 padding: .5rem 1rem;
24                 font-size: 1.5rem;
25                 text-align: center;
26                 border-bottom: 1px solid #f1f1f1;
```

#### 4. Construire la todo-list

```
27     }
28     h1 label {
29         display: block;
30         white-space: nowrap;
31         overflow: hidden;
32         text-overflow: ellipsis;
33     }
34
35     form {
36         flex: 1 0 auto;
37         display: flex;
38         flex-direction: column;
39         padding: 0;
40         text-align: center;
41         font-weight: bold;
42         align-items: center;
43         justify-content: space-between;
44     }
45     form::before {
46         content: '';
47         height: 0;
48         flex: 0 0 auto;
49     }
50     input {
51         justify-self: center;
52         flex: 0 0 auto;
53         width: 75%;
54         border-bottom: 1px solid #f1f1f1;
55     }
56     input:hover,
57     input:focus {
58         border-color: #c4c4c4;
59     }
60     ::-moz-placeholder {
61         color: #444;
62         font-style: italic;
63         text-align: center;
64     }
65     ::-webkit-input-placeholder {
66         color: #444;
67         font-style: italic;
68         text-align: center;
69     }
70     button {
71         flex: 0 0 auto;
72         width: 100%;
73         border-top: 1px solid #f1f1f1;
74     }
75 };
76
```

#### 4. Construire la todo-list

```
77
78     constructor() {
79         super()
80     }
81
82     connectedCallback() {
83         super.connectedCallback()
84
85         // On attend que le HTML soit évalué par le
            navigateur avant d'appliquer les écouteurs
86         setTimeout(() => {
87             const form = this.shadowRoot.getElementById('formulaire-creation-liste')
88
89             this.shadowRoot.addEventListener('submit',
                this.creerListe.bind(this))
90
91             this.shadowRoot.addEventListener('input',
                () => {
92                 const input =
                    this.shadowRoot.querySelector('form input[type="text"]')
93                 const btn =
                    this.shadowRoot.querySelector('form button[type="submit"]')
94
95                 btn.disabled = !(input.value &&
                    input.value.length)
96             })
97         }, 0)
98     }
99
100     creerListe(event) {
101         event.preventDefault()
102
103         const input = this.shadowRoot.getElementById('input-ajout-liste')
104
105         const task = new TodoList(input.value)
106
107         input.value = ''
108     }
109
110     render() {
111         return html`
112             <h1>
113                 <label for="input-ajout-liste">Créer une liste</label>
114             </h1>
115     `
```

#### 4. Construire la *todo-list*

```
116     <form id="formulaire-creation-liste">
117   |
118   |     <input type="text" id="input-ajout-liste" aria-inva
119   |     <button type="submit" disabled>Créer</button>
120   |   </form>`
121   }
122 window.customElements.define('new-todo-list', NewTodoListElement,
    { extends: 'section' })
```

Listing 7 – elements/new-todolist.element.js

### 4.4. Remonter les informations

Maintenant que les informations voyagent dans un sens, il serait temps de les faire passer dans l'autre sens pour les propager... avec des événements !

Par exemple, pour la création d'une liste :

```
1  const liste = new TodoList(input.value); // On crée une nouvelle
   liste
2
3  const newListEvent = new CustomEvent('new-list', {
4    bubbles: true,
5    composed: true,
6    detail: liste // On transmet notre nouvelle liste dans
   l'événement
7  })
8
9  this.dispatchEvent(newListEvent) // On transmet l'événement
```

Listing 8 – elements/new-todolist.element.js

On peut alors récupérer l'événement plus haut :

```
1  this.renderRoot.addEventListener('new-list', (event) => {
2    this.listes.push(event.detail); // On récupère la liste
   dans l'événement
3    this.update(); // On re-calcul le template pour afficher
   la nouvelle liste
4  })
```

Listing 9 – elements/app.element.js

## Conclusion

Et voilà, on a maintenant une belle *todo-list* ! 🍎

## 5. Le stockage

### Introduction

Maintenant que l'on sait faire circuler des informations dans notre application il reste un petit souci : *si on recharge la page tout est perdu !* 🍊

Voyons alors comment sauvegarder tout ça pour s'en servir plus tard !

### 5.1. Le stockage local (localStorage)

Principalement utilisé pour stocker des informations côté client, pour éviter de faire circuler des cookies inutilement sur le réseau, le `localStorage` permet d'enregistrer et lire du texte dans le navigateur grâce à une [API](#) synchrone simple.

### 5.2. Communiquer avec un serveur

### Conclusion

Et voilà, votre todo-list est sauvegardée en cas de changement et chargée automatiquement au lancement de l'application ! 🍊

# Conclusion

Et voilà, votre todo-list est prête à être utilisée !  
Vous pouvez retrouver le code complet [sur GitHub](#) ↗ .

*i*

Vous pouvez également tester [ma version en ligne](#) ↗ gratuitement et librement.

# Liste des abréviations

**API** Interface de programmation (Application Programming Interface). 26

**DOM** Document Object Model. 13, 14

**JS** JavaScript. 14

**MDN** Mozilla Developer Network. 6, 9