JS

Navigateur : Document, Évènements, Interfaces

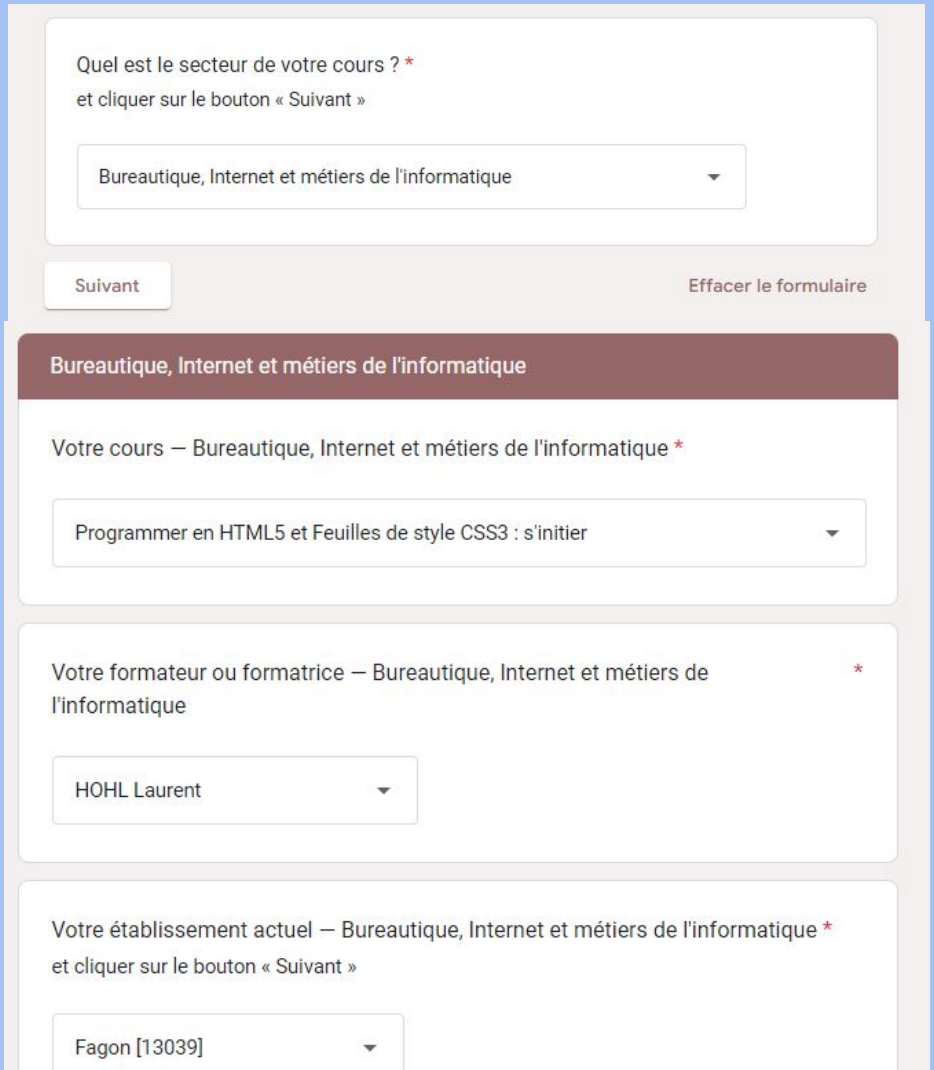
4

Dernier cours quelques infos

Evaluation du cours

Un formulaire permet d'évaluer le cours. Merci de bien vouloir le remplir :

http://aviscap.fr.nf/



Les Formulaires

Navigation : formulaire et éléments

Les formulaires des documents sont membres de la collection spéciale **document.forms.**

Il s'agit d'une collection dite "nommée" : elle est à la fois nommée et ordonnée.

Nous pouvons utiliser le nom ou le numéro du document pour obtenir le formulaire.

```
document.forms.my; // Le formulaire nommé "my"
document.forms[0]; // Le premier formulare du document
```

Lorsque nous avons un formulaire, tout élément est disponible dans la collection nommée **form.elements**.

Par exemple :

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>
<script>
  // Sélectionne le formulaire
  let form = document.forms.my; // <form name="my">
  // Sélectionne le champs
  let elem = form.elements.one; // <input name="one">
  alert(elem.value); // 1
</script>
```

Les Formulaires

Il peut y avoir plusieurs éléments portant le même nom. Ceci est typique avec les boutons radio et les cases à cocher.
Dans ce cas, form.elements[name]c'est une collection . Par exemple:

```
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>
<script>
let form = document.forms[0];
let ageElems = form.elements.age;
alert(ageElems[0]); // [object HTMLInputElement]
</script>
```

Ces propriétés de navigation ne dépendent pas de la structure des balises. Tous les éléments de contrôle, quelle que soit leur profondeur dans le formulaire, sont disponibles au format form.elements.

Les Formulaires

Ensembles de champs, fieldset, en tant que « sous-formulaires »

Un formulaire peut contenir un ou plusieurs éléments <fieldset>. Ils ont également une propriété *elements* qui répertorie les contrôles de formulaire à l'intérieur. Par exemple:

```
<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>
  <script>
    alert(form.elements.login); // <input name="login">
    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement
    // On peut référencer le champs soit par son nom soir
       par son fieldset
    alert(fieldset.elements.login == form.elements.login);
// true
  </script>
</body>
```

Les Formulaires

Notation plus courte : form.name

Il existe une notation plus courte : nous pouvons accéder à l'élément sous la forme form[index/nom].

En d'autres termes, au lieu de form.elements.login, nous pouvons écrire form.login.

Cela fonctionne également, mais il y a un petit effet : si nous accédons à un élément et que nous changeons ensuite son nom, il est toujours disponible sous l'ancien nom (ainsi que sous le nouveau).

C'est facile à voir dans un exemple :

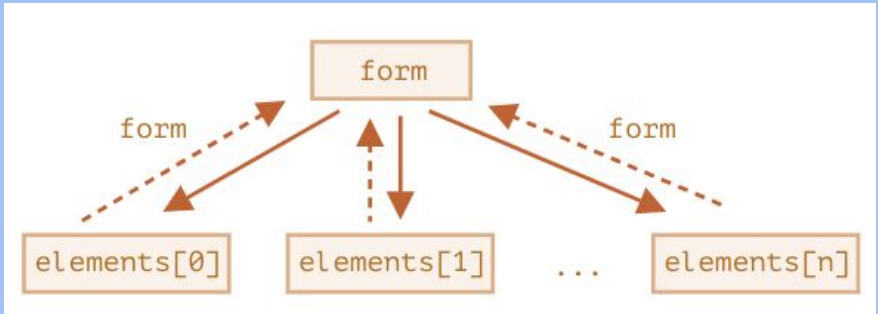
```
<body>
  <form id="form">
     <input name="login" type="text">
</form>
<script>
  alert(form.elements.login == form.login ); // true
  form.login.name=”username”
  // form.elements mis à jour:
  alert(form.elements.login);    // undefined
  alert(form.elements.username); // input

  // form autorise les deux noms
  alert(form.username == form.login); // true
</script>
</body>
```

Ce n'est généralement pas un problème, car nous modifions rarement les noms des éléments du formulaire.

Les Formulaires

Référence arrière : element.form

Pour tout élément, le formulaire est disponible sous la forme element.form. Ainsi, un formulaire fait référence à tous les éléments, et les éléments font référence au formulaire.




```
<form id="form">
     <input name="login" type="text">
</form>
<script>
 let login = form.login; // form -> element form.elements mis à jour:
  alert( login.form );  // HTMLFormElement
</script>
```

Les Formulaires

Éléments de formulaire

Parlons des contrôles de formulaires.

input et textarea

Nous pouvons accéder à leur valeur sous la forme input.value (chaîne de caractères) ou input.checked (booléen) pour les cases à cocher et les boutons radio. Voici ce que cela donne :

```
input.value = "New value";
textarea.value = "New text";

input.checked = true; // for a checkbox or radio button
```

Utilisez textarea.value, pas textarea.innerHTML

Veuillez noter que même si <textarea>...</textarea> conserve sa valeur de HTML imbriqué, nous ne devrions jamais utiliser textarea.innerHTML pour y accéder. Il stocke uniquement le HTML initialement présent sur la page, pas la valeur actuelle.

Les Formulaires

Select et option

Un élément select possède trois propriétés importantes :

1. select.options - la collection des sous-éléments de l'option,
2. select.value - la valeur de l'option sélectionnée,
3. select.selectedIndex - le numéro de l'option actuellement sélectionnée.

Ces propriétés permettent de définir la valeur d'un select de trois manières différentes :

1. Trouver l'élément d'option correspondant (par exemple parmi select.options) et fixer son option.selected à true.
2. Si nous connaissons une nouvelle valeur : fixer select.value à la nouvelle valeur.
3. Si nous connaissons le nouveau numéro de l'option : fixer select.selectedIndex à ce numéro.

Les Formulaires

Voici un exemple de ces trois méthodes :

```
<select id="select">
  <option value="apple">Pomme</option>
  <option value="pear">Poire</option>
  <option value="banana">Banane</option>
</select>
<script>
  // Les trois lignes ont le même effet
  select.options[2].selected = true;
  select.selectedIndex = 2;
  select.value = 'banana';
  // Remarque : les options commencent à zéro,
     l'indice 2 correspond donc à la troisième option.
</script>
```

Contrairement à la plupart des autres contrôles, **<select>** permet de sélectionner plusieurs options à la fois s'il possède l'attribut **multiple**. Cet attribut est cependant rarement utilisé. Pour les valeurs sélectionnées multiples, utilisez la première méthode de définition des valeurs : ajoutez/supprimez la propriété sélectionnée des sous-éléments **<option>**.

Les Formulaires

Voici un exemple de récupération des valeurs sélectionnées à partir d'une sélection multiple :

```
<select id="select" multiple>
  <option value="apple" selected>Pomme</option>
  <option value="pear" selected>Poire</option>
  <option value="banana">Banane</option>
</select>
<script>
  // get all selected values from multi-select
  let selected = Array.from( select.options )
    .filter(option => option.selected)
    .map(option => option.value);
  alert(selected); // apple,pear
</script>
```

Les Formulaires

Nouvelle option

La spécification contient une syntaxe courte et agréable pour créer un élément <option> :

```
option = new Option(text, value, defaultSelected, selected);
```

Cette syntaxe est facultative. Nous pouvons utiliser **document.createElement('option')** et définir les attributs manuellement. Néanmoins, cela peut être plus court, voici donc les paramètres :

- **text** – le texte de l'option,
- **value** – la valeur de l'option,
- **defaultSelected** – si vrai, l'attribut HTML sélectionné est créé,
- **selected** – si vrai, l'option est sélectionnée.

La différence entre **defaultSelected** et **selected** est que **defaultSelected** définit l'attribut HTML (que nous pouvons obtenir en utilisant **option.getAttribute**('selected'), tandis que **selected** définit si l'option est sélectionnée ou non.

Dans la pratique, il convient généralement de définir les deux valeurs à **true** ou false. (Ou, tout simplement, les omettre ; elles prennent toutes deux la valeur **false** par défaut).

Les Formulaires

Par exemple, voici une nouvelle option "non sélectionnée" :

```
let option = new Option("Text", "value");
// creates <option value="value">Text</option>
```

La même option, sélectionnée

```
let option = new Option("Text", "value", true, true);
```

Les éléments d'option ont des propriétés :

- **option.selected** : L'option est-elle sélectionnée.
- **option.index** : Le numéro de l'option parmi les autres dans son <select>.
- **option.text** : Contenu textuel de l'option (vu par le visiteur).

Les Formulaires

Focus: focus/blur

Un élément reçois le focus quand l'utilisateur clique dessus ou utilise la touche Tab du clavier. Il y a aussi un attribut HTML autofocus qui met le focus sur un élément par défaut lorsque la page charge et d'autres moyens d'obtenir un focus sont utilisées.

Avoir le focus sur un élément signifie généralement: "préparez-vous à accepter des données ici", c'est donc le moment où nous pouvons exécuter le code pour initialiser la fonctionnalité requise.
Le moment où le focus est perdu ("blur") peut être encore plus important. C'est quand l'utilisateur clique ailleurs ou appuie sur Tab, ou d'autre moyen, pour aller au champs de formulaire suivant.

Perdre le focus signifie généralement: "la donnée a été entrée", nous pouvons donc exécuter le code pour la vérifier ou même pour la sauvegarder sur le serveur etc.

Il y a d'importantes particularités lorsque l'on travaille avec les événements de focus.

Les Formulaires

Évènements focus/blur

L'évènement **focus** est appelé lors du focus, et blur – lorsque l'élément perd le focus
Utilisons les pour la validation d'un champ de saisie.

Dans l'exemple ci-dessous:

- Le gestionnaire **blur** vérifie si l'adresse mail est entrée, et sinon – affiche une erreur.
- Le gestionnaire **focus** masque le message d'erreur (au moment de blur le champ sera vérifié à nouveau): <style>

```
<style>
  .error {
    background: red;
  }
</style>
Entrez votre email: <input type="email" id="input">
<input type="text" style="width:220px" placeholder="entrez une adresse email invalide
et essayez de mettre le focus sur ce champ">
```

Les Formulaires

```
<script>
  input.onblur = function() {
    if (!this.value.includes('@')) { // pas une adresse email
      // affiche l'erreur
      this.classList.add("error");
      // ...et remet le focus
      input.focus();
    } else {
      this.classList.remove("error");
    }
  };
</script>
```

Cela fonctionne sur tous les navigateurs à l'exception de Firefox (bug). Si nous entrons quelque chose dans le champ de saisie puis essayons de Tab ou de cliquer en dehors du <input>, alors **onblur** remet le focus. Veuillez noter que nous ne pouvons pas "empêcher la perte de focus" en appelant **event.preventDefault()** dans **onblur**, parce que **onblur** fonctionne après que l'élément ait perdu le focus.

Les Formulaires

Permettre de mettre le focus sur n'importe quel élément: tabindex

Par défaut beaucoup d'éléments ne supportent pas le focus.

La liste change un peu selon les navigateurs, mais une chose est toujours vrai: le support de focus/blur est garanti pour les éléments avec lesquels le visiteur peut interagir: <button>, <input>, <select>, <a>, etc.

D'un autre côté, les éléments qui existent pour mettre quelque chose en forme, comme , , <table> – ne peuvent pas recevoir de focus par défaut. La méthode **elem.focus()** ne fonctionne pas sur eux, et les évènements **focus/blur** ne sont jamais déclenchés.

Cela peut être changé en utilisant l'attribut HTML **tabindex**.

N'importe quel élément peut recevoir le focus s'il a **tabindex**. La valeur de l'attribut est l'ordre de l'élément lorsque Tab est utilisé pour changer d'élément.

C'est-à-dire: si nous avons deux éléments, le premier ayant **tabindex="1"**, et le deuxième ayant **tabindex="2"**, alors appuyer sur Tab en étant sur le premier élément – déplace le focus sur le deuxième.

L'ordre de changement est: les éléments avec tabindex à 1 et plus sont en premier (dans l'ordre des tabindex), puis les éléments sans tabindex.

Les Formulaires

Les éléments sans tabindex correspondant sont dans l'ordre du source du document (l'ordre par défaut)

Il y a deux valeurs spéciales:

- **tabindex="0"** met l'élément parmi ceux qui ont tabindex. C'est-à-dire, lorsque l'on change d'élément, les éléments avec tabindex=0 vont après ceux avec tabindex ≥ 1. Généralement c'est utilisé pour permettre à un élément d'obtenir le focus tout en gardant l'ordre de changement par défaut. Pour intégrer un élément dans le formulaire de la même manière qu'un <input>.
- **tabindex="-1"** permet seulement le focus par programmation. La touche Tab ignore ces éléments, mais la méthode **elem.focus()** fonctionne.

Par exemple, voici une liste. Cliquez sur le premier élément et appuyez sur Tab.

L'ordre est comme ceci: 1 - 2 - 0. Normalement, <li> ne supporte pas le focus, mais tabindex l'active, avec les évènements et le pseudo-sélecteur CSS :focus.

Les Formulaires

```
Cliquez sur le premier élément et appuyez sur Tabulation. Suivez
l'ordre. Veuillez noter que plusieurs Tabulations à la suite
peuvent déplacer le focus en dehors de l'iframe avec l'exemple.
<ul>
  <li tabindex="1">Un</li>
  <li tabindex="0">Zéro</li>
  <li tabindex="2">Deux</li>
  <li tabindex="-1">Moins un</li>
</ul>
<style>
  li { cursor: pointer; }
  :focus { outline: 1px dashed green; }
</style>
```

La propriété elem.tabIndex fonctionne aussi

Nous pouvons ajouter tabindex depuis le JavaScript en utilisant la propriété **elem.tabIndex.** Cela a le même effet.

Les Formulaires

Les événements: change, input, cut, copy, paste

Découvrons différents événements qui accompagnent les mises à jour des données.

Événement: change

L'événement **change** se déclenche lorsque le changement de la valeur de l'élément a fini de se réaliser.

Pour les éléments input de type text cela signifie que l'événement se produit lorsqu'ils perdent le focus.

Par exemple, pendant que nous saisissons dans le champ de texte ci-dessous – il n'y a aucun événement. Mais lorsque nous déplaçons le focus ailleurs, par exemple, en cliquant sur un bouton – il y aura un événement change:

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">
```

Les Formulaires

Pour les autres éléments: select, input type=checkbox/radio il se déclenche juste après les changements de sélection:

```
<select onchange="alert(this.value)">
  <option value="">Choisissez</option>
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
  <option value="3">Option 3</option>
</select>
```

Événement: input

L'événement **input** se déclenche à chaque fois qu'une valeur est modifiée par l'utilisateur.

Contrairement aux événements liés au clavier, il se déclenche sur toute modification de valeur, même celles qui n'impliquent pas d'actions du clavier: coller avec une souris ou utiliser la reconnaissance vocale pour dicter le texte.

Par exemple:

Les Formulaires

```
<input type="text" id="input"> oninput: <span id="result"></span>
<script>
  input.oninput = function() {
    result.innerHTML = input.value;
  };
</script>
```

Événements: cut, copy, paste

Ces événements se produisent lors de la coupe/copie/collage d'une valeur.

Ils appartiennent à la classe ClipboardEvent et permettent d'accéder aux données coupées/copiées/collées.

Nous pouvons également utiliser event.preventDefault() pour interrompre l'action, puis rien n'est copié/collé.

Par exemple, le code ci-dessous empêche tous ces événements cut/copy/paste et montre ce que nous essayons de couper/copier/coller:

Les Formulaires

```
<input type="text" id="input">
<script>
  input.onpaste = function(event) {
    alert("paste: " + event.clipboardData.getData('text/plain'));
    event.preventDefault();
  };
  input.oncut = input.oncopy = function(event) {
    alert(event.type + '-' + document.getSelection());
    event.preventDefault();
  };
</script>
```

Remarque : à l'intérieur des gestionnaires d'événements cut et copy, un appel à event.clipboardData.getData(...) renvoie une chaîne vide. C'est parce que techniquement, les données ne sont pas encore dans le presse-papiers. Si nous utilisons event.preventDefault(), il ne sera pas du tout copié.

Ainsi, l'exemple ci-dessus utilise document.getSelection() pour obtenir le texte sélectionné.

Les Formulaires

Il est possible de copier/coller pas seulement du texte, mais tout. Par exemple, nous pouvons copier un fichier dans le gestionnaire de fichiers du système d'exploitation et le coller.

C'est parce que **clipboardData** implémente l'interface DataTransfer, couramment utilisée pour le glisser-déposer et le copier/coller. C'est un peu hors de notre portée maintenant, mais vous pouvez trouver ses méthodes dans la spécification DataTransfer.

En outre, il existe une API asynchrone supplémentaire pour accéder au presse-papiers : navigator.clipboard. Plus d'informations à ce sujet dans la spécification **Clipboard API and events**, non pris en charge par Firefox.

Restrictions de sécurité

Le presse-papiers est une chose "globale" au niveau du système d'exploitation. Un utilisateur peut basculer entre différentes applications, copier/coller différentes choses, et une page de navigateur ne devrait pas voir tout cela. Ainsi, la plupart des navigateurs permettent un accès en lecture/écriture transparent au presse-papiers uniquement dans le cadre de certaines actions de l'utilisateur, telles que le copier/coller, etc.

Les Formulaires

Il est interdit de générer des événements de presse-papiers "personnalisés" avec **dispatchEvent** dans tous les navigateurs sauf Firefox. Et même si nous parvenons à envoyer un tel événement, la spécification indique clairement que de tels événements "synthétiques" ne doivent pas donner accès au presse-papiers. Même si quelqu'un décide d'enregistrer **event.clipboardData** dans un gestionnaire d'événements, puis d'y accéder plus tard, cela ne fonctionnera pas.

Pour réitérer, **event.clipboardData** fonctionne uniquement dans le contexte des gestionnaires d' événements initiés par l'utilisateur.

D'autre part, **navigator.clipboard** est l'API la plus récente, destinée à être utilisée dans n'importe quel contexte. Elle demande l'autorisation de l'utilisateur, si nécessaire.

Les Formulaires

Formulaires: l'événement et la méthode "submit"

L'événement **submit** se déclenche lorsque le formulaire est soumis, il est généralement utilisé pour valider le formulaire avant de l'envoyer au serveur ou pour abandonner la soumission et la traiter en JavaScript.

La méthode **form.submit()** permet de lancer l'envoi de formulaire depuis JavaScript. Nous pouvons l'utiliser pour créer et envoyer dynamiquement nos propres formulaires au serveur.

Voyons-les plus en détail.

Évènement: submit

- Le premier – cliquer sur <input type="submit"> ou <input type="image">.
- La seconde – appuyez sur Enter dans un champ de saisie.

Les deux actions mènent à l'événement submit sur le formulaire. Le gestionnaire peut vérifier les données, et s'il y a des erreurs, les afficher et appeler event.preventDefault(), alors le formulaire ne sera pas envoyé au serveur.

Les Formulaires

Dans le formulaire ci-dessous:

1. Allez dans le champ de texte et appuyez sur Entrée.
2. Cliquez sur <input type ="submit">.

Les deux actions affichent alert et le formulaire n'est envoyé nulle part en raison de *return false*:

```
<form onsubmit="alert('submit!');return false">
  Premièrement : Entrer dans le champ de saisie <input type="text" value="text"><br>
  Deuxièment: Cliquer sur "submit": <input type="submit" value="Submit">
</form>
```

Méthode: submit

Pour soumettre manuellement un formulaire au serveur, nous pouvons appeler **form.submit().**

Ensuite, l'événement **submit** n'est pas généré. On suppose que si le programmeur appelle **form.submit(),** alors le script a déjà effectué tous les traitements associés.

Les Formulaires

Parfois, cela est utilisé pour créer et envoyer manuellement un formulaire, comme ceci:

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// le formulaire doit être dans le document pour le soumettre
document.body.append(form);

form.submit();
```

Les évènements

Chargement du document et des ressources

Page: DOMContentLoaded, load, beforeunload, unload

Le cycle de vie d'une page HTML comporte trois événements importants:

- **DOMContentLoaded** – le navigateur a complètement chargé le HTML et l'arborescence DOM est construite, mais les ressources externes telles que les images <img> et les feuilles de style peuvent ne pas être encore chargées.
- **load** – non seulement le HTML est chargé, mais également toutes les ressources externes : images, styles, etc.
- **beforeunload/unload** – l'utilisateur quitte la page.

Les évènements

Chaque événement peut être utile:

- **DOMContentLoaded événement** – DOM est prêt, le gestionnaire peut donc rechercher des nœuds DOM, initialiser l'interface.
- **load événement** – les ressources externes sont chargées, donc les styles sont appliqués, les tailles d'image sont connues, etc.
- **beforeunload** – l'utilisateur quitte la page, nous pouvons vérifier si l'utilisateur a enregistré les modifications et leur demander s'ils veulent vraiment partir.
- **unload** – l'utilisateur est presque parti, mais nous pouvons toujours lancer certaines opérations, comme l'envoi de statistiques.

Explorons les détails de ces événements.

DOMContentLoaded

L'événement DOMContentLoaded se produit sur l'objet document. Nous devons utiliser **addEventListener** pour le capturer:

```
document.addEventListener("DOMContentLoaded", ready);
// pas "document.onDOMContentLoaded = ..."
```

Les évènements

Par exemple:

```
<script>
  function ready() {
    alert('DOM is ready');
    // l'image n'est pas encore chargée (sauf si elle a été mise en cache), donc la taille est 0x0
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  }
  document.addEventListener("DOMContentLoaded", ready);
</script>
<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

Dans l'exemple, le gestionnaire DOMContentLoaded s'exécute lorsque le document est chargé, afin qu'il puisse voir tous les éléments, y compris <img> dessous.

Mais il n'attend pas que l'image se charge. Ainsi, alert n'affiche aucune taille.

À première vue, l'événement DOMContentLoaded est très simple. L'arbre DOM est prêt – voici l'événement. Il y a cependant quelques particularités.

Les évènements

DOMContentLoaded et les scripts

Lorsque le navigateur traite un document HTML et rencontre une balise **<script>**, il doit s'exécuter avant de continuer à construire le DOM. C'est une précaution, car les scripts peuvent vouloir modifier DOM, avec document.write par exemple, donc **DOMContentLoaded** doit attendre. Donc **DOMContentLoaded** se produit définitivement après de tels scripts:

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    alert("DOM ready!");
  });
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>
<script>
alert("Librairie chargée et script inline exécuté!");
</script>
```

Les évènements

Les scripts qui ne bloquent pas DOMContentLoaded

Il existe deux exceptions à cette règle:

1. Les scripts avec l'attribut **async**, que nous aborderons un peu plus tard, ne bloquent pas DOMContentLoaded.
2. Les scripts qui sont générés dynamiquement avec **document.createElement(**'script') puis ajoutés à la page Web ne bloquent pas non plus cet événement.

DOMContentLoaded et les styles

Les feuilles de style externes n'affectent pas le DOM, donc **DOMContentLoaded** ne les attend pas.

Mais il y a une embûche. Si nous avons un script après le style, ce script doit attendre que la feuille de style se charge:

Les évènements

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // le script ne s'exécute pas tant que la feuille de style n'est pas chargée
  alert(getComputedStyle(document.body).marginTop);
</script>
```

La raison en est que le script peut souhaiter obtenir des coordonnées et d'autres propriétés d'éléments dépendant du style, comme dans l'exemple ci-dessus. Naturellement, il doit attendre le chargement des styles.

Comme **DOMContentLoaded** attend les scripts, il attend maintenant les styles avant eux également.

Saisie automatique intégrée du navigateur

Firefox, Chrome et Opera remplissent automatiquement les formulaires sur **DOMContentLoaded**.

Par exemple, si la page a un formulaire avec login et mot de passe, et que le navigateur se souvient des valeurs, alors sur **DOMContentLoaded**, il peut essayer de les remplir automatiquement (si approuvé par l'utilisateur).

Les évènements

Donc, si **DOMContentLoaded** est reporté par des scripts à chargement long, le remplissage automatique attend également. Vous l'avez probablement vu sur certains sites (si vous utilisez le remplissage automatique du navigateur) – les champs de login/mot de passe ne sont pas remplis automatiquement immédiatement, mais il y a un délai jusqu'à ce que la page se charge complètement.

C'est en fait le délai jusqu'à l'événement **DOMContentLoaded**.

window.onload

L'événement **load** sur l'objet window se déclenche lorsque la page entière est chargée, y compris les styles, images et autres ressources.

Cet événement est disponible via la propriété **onload**.

L'exemple ci-dessous montre correctement les tailles d'image, car **window.onload** attend toutes les images:

Les Formulaires

```
<script>
 window.onload = function(){ // peut utiliser window.addEventListener('load', (event) => {
    alert('Page chargée);
    alert(`Image: ${img.offsetWidth}x${img.offsetHeight}`);
  }
</script>
<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

window.onunload

Lorsqu'un visiteur quitte la page, l'événement **unload** se déclenche sur **window**. Nous pouvons y faire quelque chose qui n'implique pas de retard, comme la fermeture des fenêtres contextuelles associées.

L'exception notable est l'envoi d'analyses. Disons que nous recueillons des données sur la façon dont la page est utilisée: clics de souris, défilements, zones de page visualisées, etc.

Naturellement, l'événement **unload** est lorsque l'utilisateur nous quitte, et nous aimerions sauvegarder les données sur notre serveur.

Les évènements

Il existe une méthode spéciale navigator.**sendBeacon**(url, data) pour de tels besoins, décrite dans la spécification https://w3c.github.io/beacon/.

Il envoie les données en arrière-plan. La transition vers une autre page n'est pas retardée: le navigateur quitte la page, mais exécute toujours **sendBeacon**.

Voici comment l'utiliser:

```
let analyticsData = { /* objet avec des données collectées */ };
window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
});
```

Lorsque la demande **sendBeacon** est terminée, le navigateur a probablement déjà quitté le document, donc il n'y a aucun moyen d'obtenir une réponse du serveur (qui est vide habituellement pour l'analyse).

Si nous voulons annuler la transition vers une autre page, nous ne pouvons pas le faire ici.

Mais nous pouvons utiliser un autre événement – **onbeforeunload**.

Les évènements

window.onbeforeunload

Si un visiteur a lancé la navigation hors de la page ou tente de fermer la fenêtre, le gestionnaire beforeunload demande une confirmation supplémentaire.

Si nous annulons l'événement, le navigateur peut demander au visiteur s'il en est sûr.

Vous pouvez l'essayer en exécutant ce code, puis en rechargeant la page:

```
window.onbeforeunload = function() {
  return false;
};
```

Pour des raisons historiques, renvoyer une chaîne non vide compte également comme une annulation de l'événement.

Voici un exemple:

Les évènements

```
window.onbeforeunload = function() {
  return "Il y a des modifications, partir quand même ?";
};
```

Le comportement a été modifié, car certains webmasters ont abusé de ce gestionnaire d'événements en affichant des messages trompeurs et ennuyeux. Donc, à l'heure actuelle, les anciens navigateurs peuvent toujours l'afficher sous forme de message, mais à part cela – il n'y a aucun moyen de personnaliser le message affiché à l'utilisateur.

readyState

Que se passe-t-il si nous définissons le gestionnaire **DOMContentLoaded** après le chargement du document ?

Naturellement, il ne fonctionne jamais. Il y a des cas où nous ne savons pas si le document est prêt ou non. Nous aimerions que notre fonction s'exécute lorsque le DOM est chargé, que ce soit maintenant ou plus tard.

La propriété document.**readyState** nous renseigne sur l'état de chargement actuel.

Les évènements

Il y a 3 valeurs possibles:

- "**loading**" – le document est en cours de chargement.
- "**interactif**" – le document a été entièrement lu.
- "**complete**" – le document a été entièrement lu et toutes les ressources (comme les images) sont également chargées.

Nous pouvons donc vérifier **document.readyState** et configurer un gestionnaire ou exécuter le code immédiatement s'il est prêt.

Comme ceci:

```
function work() { /*...*/ }
if (document.readyState == 'loading') {
  // en cours de chargement, attendez l'événement
  document.addEventListener('DOMContentLoaded', work);
} else {
  // DOM est prêt!
  work();
}
```

Les évènements

Il y a aussi l'événement **readystatechange** qui se déclenche lorsque l'état change, nous pouvons donc afficher tous ces états comme ceci:

```
// état actuel
console.log(document.readyState);
// afficher les changements d'état
document.addEventListener('readystatechange', () => console.log(document.readyState));
```

L'événement **readystatechange** est un mécanisme alternatif de suivi de l'état de chargement du document, il est apparu il y a longtemps.

De nos jours, il est rarement utilisé.

Voyons le flux complet des événements pour l'exhaustivité.

Voici un document avec **<iframe>**, **<img>** et des gestionnaires qui consignent les événements:

Les évènements

```
<script>
  log('initialise readyState:' + document.readyState);
  document.addEventListener('readystatechange', () => log('readyState:'+ document.readyState));
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));
  window.onload = () => log('window onload');
</script>
<iframe src="iframe.html" onload="log('iframe onload')"></iframe>
<img src="https://en.js.cx/clipart/train.gif" id="img">
<script>
  img.onload = () => log('img onload');
</script>
```

La sortie typique:

1. [1] readyState:loading initiale
2. [2] readyState:interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState:complete
7. [4] window onload

Les nombres entre crochets indiquent l'heure approximative à laquelle cela se produit. Les événements étiquetés avec le même chiffre se produisent à peu près au même moment (± quelques ms).

- document.readyState devient interactive juste avant DOMContentLoaded. Ces deux choses signifient en fait la même chose.
- document.readyState devient complete lorsque toutes les ressources (iframe et img) sont chargées. Ici, nous pouvons voir que cela se produit à peu près en même temps que img.onload (img est la dernière ressource) et window.onload. Passer à l'état complete signifie la même chose que window.onload. La différence est que window.onload fonctionne toujours après tous les autres gestionnaires de load.

Chargement du document et des ressources

Les scripts: async, defer

Dans les sites Web modernes, les scripts sont souvent "plus lourds" que le HTML: leur taille de téléchargement est plus grande et le temps de traitement est également plus long.

Lorsque le navigateur charge le HTML et rencontre une balise **<script>...</script>**, il ne peut pas continuer à construire le DOM. Il doit exécuter le script de suite. Il en va de même pour les scripts externes **<script src ="..."></script>**: le navigateur doit attendre le téléchargement du script, l'exécuter, puis traiter le reste de la page.

Cela conduit à deux problèmes importants:

1. Les scripts ne peuvent pas voir les éléments DOM en dessous d'eux, ils ne peuvent donc pas ajouter de gestionnaires, etc.
2. S'il y a un script volumineux en haut de la page, il "bloque la page". Les utilisateurs ne peuvent pas voir le contenu de la page tant qu'il n'est pas téléchargé et exécuté:

```
<p>...contenu avant le script...</p>
<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
<!-- Ceci n'est pas visible tant que le script n'est pas chargé -->
<p>...contenu après le script...</p>
```

Chargement du document et des ressources

Il existe quelques solutions pour contourner cela. Par exemple, nous pouvons mettre un script en bas de page. Comme ça, il peut voir les éléments au-dessus, et cela ne bloque pas l'affichage du contenu de la page:

```
<body>
  ...tout le contenu au dessus du script...
<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
</body>
```

Mais cette solution est loin d'être parfaite. Par exemple, le navigateur remarque le script (et peut commencer à le télécharger) uniquement après avoir téléchargé le document HTML complet. Pour les longs documents HTML, cela peut être un retard notable.

De telles choses sont invisibles pour les personnes utilisant des connexions très rapides, mais de nombreuses personnes dans le monde ont encore des vitesses Internet lentes et utilisent une connexion Internet mobile loin d'être parfaite.

Heureusement, il y a deux attributs de **<script>** qui résolvent le problème pour nous: **defer** et **async**.

Chargement du document et des ressources

defer

L'attribut **defer** indique au navigateur de ne pas attendre le script. Au lieu de cela, le navigateur continuera à traiter le HTML, à construire le DOM. Le script se charge "en arrière-plan", puis s'exécute lorsque le DOM est entièrement construit.

Voici le même exemple que ci-dessus, mais avec **defer** :

```
<p>...contenu avant le script...</p>
<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
<!-- Ceci n'est pas visible tant que le script n'est pas chargé -->
<p>...contenu après le script...</p>
```

- Les scripts avec defer ne bloquent jamais la page.
- Les scripts avec defer s'exécutent toujours lorsque le DOM est prêt, mais avant l'événement DOMContentLoaded.

L'exemple suivant montre que:

```
<p>...contenu avant le script...</p>
<script>document.addEventListener('DOMContentLoaded',()=>alert("DOM ready après defer!")); // (2)</script>
<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
<!-- Ceci n'est pas visible tant que le script n'est pas chargé -->
<p>...contenu après le script...</p>
```

Chargement du document et des ressources

1. Le contenu de la page s'affiche immédiatement.
2. **DOMContentLoaded** attend le script différé. Il ne se déclenche que lorsque le script (2) est téléchargé et exécuté.

Les scripts différés conservent leur ordre relatif, tout comme les scripts classiques.

Donc, si nous avons d'abord un long script, puis un plus petit, alors ce dernier attend.

```
<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>
<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>
```

Les navigateurs analysent la page à la recherche de scripts et les téléchargent en parallèle pour améliorer les performances. Ainsi, dans l'exemple ci-dessus, les deux scripts se téléchargent en parallèle. Le small.js se termine probablement en premier.

... Mais l'attribut **defer**, en plus de dire au navigateur "de ne pas bloquer", garantit que l'ordre relatif est conservé. Ainsi, même si small.js se charge en premier, il attend et s'exécute toujours après l'exécution de long.js. Mais la spécification exige que les scripts s'exécutent dans l'ordre des documents, donc elle attend que long.js s'exécute.

L'attribut defer est réservé aux scripts externes
*L'attribut **defer** est ignoré si la balise <script> n'a pas de **src**.*

Chargement du document et des ressources

async

- Le navigateur ne bloque pas les scripts **async** (comme **defer**).
- D'autres scripts n'attendent pas les scripts **async**, et les scripts **async** ne les attendent pas.
- **DOMContentLoaded** et les scripts asynchrones ne s'attendent pas :
  - **DOMContentLoaded** peut se produire à la fois avant un script asynchrone (si un script **async** termine le chargement une fois la page terminée)
  - ... ou après un script **async** (si un script **async** est court ou était dans le cache HTTP)

En d'autres termes, les scripts **async** se chargent en arrière-plan et s'exécutent lorsqu'ils sont prêts. Le DOM et les autres scripts ne les attendent pas, et ils n'attendent rien. Un script entièrement indépendant qui s'exécute lorsqu'il est chargé. Aussi simple que cela puisse être, non ?

Donc, si nous avons plusieurs scripts **async**, ils peuvent s'exécuter dans n'importe quel ordre. Premier chargé – premier exécuté:

```
<p>...contenu avant le script...</p>
<script>document.addEventListener('DOMContentLoaded', () => alert("DOM ready !"));</script>
<script async src="https://javascript.info/article/script-async-defer/long.js"></script>
<script async src="https://javascript.info/article/script-async-defer/small.js"></script>
<p>...contenu après le script...</p>
```

Chargement du document et des ressources

Le contenu de la page apparaît immédiatement: **async** ne la bloque pas.

DOMContentLoaded peut arriver soit avant ou après **async**, aucune garantie ici.

Les scripts asynchrones n'attendent pas les uns les autres. Un script plus petit **small.js** passe en second, mais se charge probablement avant **long.js**, donc s'exécute en premier. C'est ce qu'on appelle une commande "load-first".

Les scripts asynchrones sont parfaits lorsque nous intégrons un script tiers indépendant dans la page: compteurs, publicités, etc., car ils ne dépendent pas de nos scripts et nos scripts ne doivent pas les attendre:

```
<!-- Google Analytics est généralement ajouté comme ceci -->
<script async src="https://google-analytics.com/analytics.js"></script>
```

L'attribut async est réservé aux scripts externes
L'attribut async est ignoré si la balise <script> n'a pas de src.

Les scripts dynamiques
Il existe un autre moyen important d'ajouter un script à la page.
Nous pouvons également ajouter un script dynamiquement en utilisant JavaScript:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
script.async = false;
document.body.append(script);
```

Chargement du document et des ressources

Par exemple, nous ajoutons ici deux scripts.

Sans **script.async=false**, ils s'exécuteraient dans l'ordre de chargement (le small.js probablement en premier). Mais avec, l'ordre est "comme dans le document":

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.body.append(script);
}

// long.js s'exécute en premier à cause de async=false
loadScript("/article/script-async-defer/long.js");
loadScript("/article/script-async-defer/small.js");
```

Stockage des données dans le navigateur

Cookies, document.cookie

Les cookies sont des donnés stockées sous forme de petites chaîne de caractères directement dans le navigateur. Ils font parti du protocole HTTP, ils sont définis par la spécification RFC 6265.

Les cookies sont en général définis par le serveur web en utilisant l'entête HTTP Set-Cookie. Alors, le navigateur les ajoutent automatiquement à (presque) toutes les requêtes provenant du même domaine en utilisant l'entête HTTP Cookie.

L'un des cas d'utilisation les plus répandus est l'authentification :

1. Une fois connecté, le serveur utilise l'entête HTTP Set-Cookie dans la réponse pour définir un cookie avec un "identifiant de session" unique.
2. La prochaine fois lorsque la requête est envoyée au même domaine, le navigateur envoie le cookie sur le réseau en utilisant l'entête HTTP Cookie.
3. Alors le serveur sait qui a fait la requête.

Nous pouvons aussi accéder aux cookies depuis le navigateur, en utilisant la propriété document.cookie.

Il y a beaucoup de chose malignes à faire à propos des cookies et leurs options.

Stockage des données dans le navigateur

Lire depuis document.cookie

Votre navigateur stocke t-il des cookies depuis le site Paris:fr ? Voyons voir :

```
// Sur paris.fr, il devrait y avoir quelques cookies
// Aller dans la console et taper
alert( document.cookie ); // Des infos apparaissent
```

La valeur de **document.cookie** consiste en des paires name=value, délimitées par :. Chacune étant un cookie séparé.

Pour trouver un cookie en particulier, nous pouvons diviser **document.cookie** par :, et donc trouver le bon nom. Nous pouvons utiliser soit une expression régulière (regex) ou des fonctions de tableau pour faire cela.

Nous laissons cela en tant qu'exercice pour le lecteur.

A la fin du chapitre vous trouverez des fonctions utilitaires pour manipuler les cookies.

Stockage des données dans le navigateur

Écrire dans document.cookie

Nous pouvons écrire dans **document.cookie**. Mais ce n'est pas une propriété de données, c'est un accesseur (getter/setter). Une affectation à ce dernier est traitée spécialement.

Une opération d'écriture dans **document.cookie** met à jour seulement les cookies mentionnés dedans, mais ne touche pas les autres cookies.

Par exemple, cet appel définit un cookie avec le nom **user** et la valeur **John** :

```
document.cookie = "user=John"; // Met à jour uniquement le cookie nommé 'user'
alert( document.cookie ); // Affiche tous les cookies
```

Si vous exécutez cela, vous verrez probablement plusieurs cookies. Car l'opération document.cookie= ne réécrit pas tous les cookies. Elle définit uniquement le cookie **user** mentionné.

Techniquement, le nom et la valeur peuvent être n'importe quel caractère. Pour garder le formattage valide, ils devraient être échappés en utilisant la fonction integrée **encodeURIComponent** :

Stockage des données dans le navigateur

```
// Les caractères spéciaux ont besoin d'encodage
let name = "my name";
let value = "John Smith"

// Encode le cookie en tant que my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert( document.cookie ); // ...; my%20name=John%20Smith
```

Limitations

Il y a quelques limites :

La paire name=value, après encodeURIComponent, ne peut pas excéder 4KB. Donc on ne peut pas stocker quelque chose de trop lourd sur un cookie.

Le nombre total de cookie par domaine est limité à ~ 20+, la limite exacte dépend du navigateur.

Les cookies ont plusieurs options, beaucoup d'entre elles sont importantes et devraient être définies.

Les options sont listées après "*key=value*", délimité par "*;*", comme ceci :

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

Stockage des données dans le navigateur

path

- *path=/mypath*

Le préfix du chemin de l'URL doit être absolu. Ça rend le cookie accessible pour les pages du même chemin. Par défaut, il s'agit du chemin courant.

Si un cookie est défini avec path=/admin, il est visible aux pages /admin et /admin/something, mais pas à /home ou /adminpage.

Généralement, nous devons définir path à la racine path=/ pour rendre le cookie accessible depuis toutes les pages du site.

domain

- *domain=site.com*

Un domaine définit par où le cookie est accessible. Cependant en pratique, il y a des limites. Nous ne pouvons pas définir n'importe quel domaine.

Stockage des données dans le navigateur

Il n'y a pas de moyen de laisser un cookie être accessible depuis un domaine de second niveau, donc other.com ne recevra jamais un cookie défini à **site.com**

Il s'agit d'une restriction de sécurité, pour nous permettre de stocker des données sensibles dans nos cookies qui ne seront disponibles que sur un site.

Par défaut, un cookie est accessible uniquement depuis le domaine qui l'a défini.

Veuillez noter, par défaut un cookie n'est pas partagé avec un sous-domaine, tel que *forum.***site.com**.

```
// Si nous définissons un cookie sur site.com
document.cookie = "user=John"

// ...Nous ne le verrons pas depuis forum.site.com
alert(document.cookie); // no user
```

...Mais cela peut changer. Si nous voulons permettre aux sous-domaines comme **forum.site.com** de récupérer un cookie défini par **site.com**, c'est possible.

Pour que cela arrive, quand nous définissons un cookie depuis **site.com,** nous pouvons définir l'option **domain** à la racine du domaine : **domain=site.com**. Alors tous les sous-domaines verront un tel cookie.

Stockage des données dans le navigateur

Par exemple :

```
// Depuis site.com
// Rendre le cookie accessible à tous les sous-domaines *.site.com:
document.cookie = "user=John; domain=site.com"

// Plus tard
// Depuis forum.site.com
alert(document.cookie); // Le cookie user=John existe
```

Pour des raisons historiques, domain=.site.con (avec un point avant site.com) fonctionne de la même manière, permettant l'accés au cookie depuis les sous-domaines. C'est une vielle notation et devrait être utilisée si nous avons besoin de prendre en charge les très vieux navigateurs.

Pour résumer, l'option domain permet de rendre un cookie accessible aux sous-domaines.

expires, max-age

Par défaut, si un cookie n'a pas ces options, il disparait quand le navigateur est fermé. De tels cookies sont appellés "cookies de session"

Stockage des données dans le navigateur

Pour laisser les cookies survivre à la fermeture du navigateur, nous pouvons soit définir soit l'option expires ou max-age.

- expires=Tue, 19 Jan 2038 03:14:07 GMT

La date d'expiration du cookie définit la date, à laquelle le navigateur le supprimera automatiquement.

La date doit être exactement dans ce format, en timezone GMT. Nous pouvons utiliser date.toUTCString pour le récupérer. Par exemple, nous pouvons définir le cookie pour qu'il expire dans 1 jour :

```
// +1 jour depuis maintenant
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

Si nous définissons expires à une date dans le passé, le cookie est supprimé.

- max-age=3600

Il s'agit d'une alternative à expires et elle spécifie l'expiration du cookie en seconde à partir de l'instant.

Stockage des données dans le navigateur

Si elle est définie à zero ou une valeur négative, le cookie sera supprimé :

```
// Le cookie mourra dans +1 heure à partir de maintenant
document.cookie = "user=John; max-age=3600";

// Supprime le cookie (le laisser expirer tout de suite)
document.cookie = "user=John; max-age=0";
```

secure

- secure

Le cookie devrait être transféré seulement avec HTTPS.

Par défaut, si nous définissons un cookie à http://site.com, alors il apparaitra aussi à https://site.com et vice versa.

Les cookies sont "domain-based", ils ne sont pas distinguables entre les protocoles.

Avec cette option, si un cookie est défini par https://site.com, alors il n'apparait pas quand le même site est accédé par HTTP, comme http://site.com. Donc si un cookie a un contenu sensible il ne devrait pas être envoyé sur HTTP qui n'est chiffré, le flag secure est la bonne chose.

```
// Considérons que nous soyons sur https:// maintenant
// Définit le cookie pour être sécurisé (seulement accessible par HTTPS)
document.cookie = "user=John; secure";
```

Stockage des données dans le navigateur

samesite

Il s'agit d'un nouvel attribut de sécurité samesite. Il a été conçu pour protéger de ce qu'on appelle attaques XSRF (cross-site request forgery).

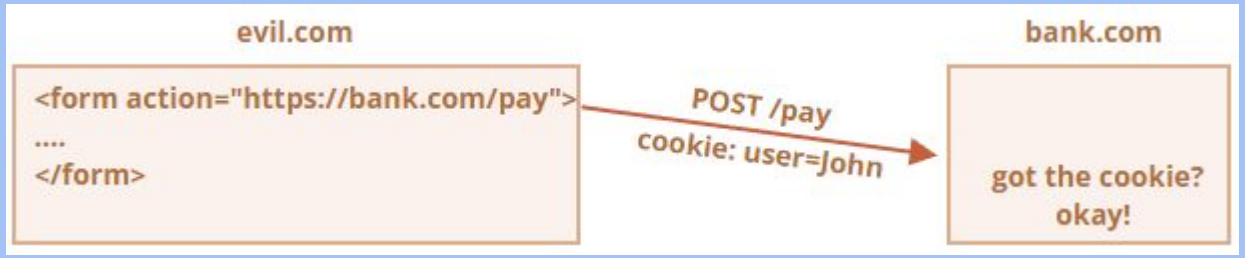
L'attaque XSRF

Imaginez, vous êtes connecté sur le site bank.com. Ce qui signifie : que vous avez un cookie d'authentification sur ce site. Votre navigateur l'envoie à bank.com à chaque requête, donc il vous reconnait et effectue toutes les opérations financières sensibles.

Maintenant, pendant que vous naviguez sur le web dans une autre fenêtre, vous arrivez accidentellement sur un autre site evil.com. Ce site a du code JavaScript qui soumet un formulaire <form action="https://bank.com/pay"> à bank.com avec les champs qui initient une transaction avec le compte du hacker.

Le navigateur envoie des cookies à chaque fois que vous visitez le site bank.com, même si le formulaire a été envoyé depuis evil.com. Donc la banque vous reconnait et effectue le paiement.

Stockage des données dans le navigateur



C'est ce qu'on appelle une attaque "Cross-Site Request Forgery" (XSRF en plus court).

Les vraies banques en sont évidemment protégées. Tous les formulaires générés par bank.com ont un champ spécial, un certain "XSRF protection token", qu'une page malveillante ne peut pas générer ou extraire de la page distante. Elle peut y soumettre un formulaire, mais pas récupérer les données. Le site bank.com vérifie ce genre de token dans tous les formulaires qu'il reçoit.

Une telle protection prend du temps à mettre en œuvre cependant. Nous avons besoin de nous assurer que tous les formulaires ont le champ de token requis, et nous devons aussi vérifier toutes les requêtes.

Entrer un cookie avec l'option samesite

L'option samesite de cookie fournit un autre moyen de se protéger de telles attaques, cela ne devrait (en théorie) pas nécessiter de "tokens de protections xsrf".

Elle a deux valeurs possible : samesite=strict (pareil que samesite sans valeur)

Stockage des données dans le navigateur

Elle a deux valeurs possible :

- samesite=strict (pareil que samesite sans valeur)

Un cookie avec samesite=strict n'est jamais envoyé si un utilisateur vient d'en dehors du même site.

En d'autres termes, qu'importe que l'utilisateur suive un lien de ses mails ou soumette un formulaire provenant d'evil.com, ou qu'il fasse des opérations provenants d'un autre domaine, le cookie n'est pas envoyé.

Si le cookie d'authentification a l'option samesite, alors l'attaque XSRF n'a aucune chance de se solder par un succés, car une soumission depuis evil.com ne vient pas avec les cookies. Donc bank.com ne reconnaitra pas l'utilisateur et ne procédera pas au paiement.

La protection est plutôt fiable. Seules les opérations provenants de bank.com vont envoyer le cookie samesite, e.g. une soumission de formulaire depuis une autre page à bank.com.

Bien que, il y ait un petit inconvénient.

Quand un utilisateur suit un lien légitime vers bank.com, comme depuis ses propres notes, il sera surpris que bank.com ne le reconnaisse pas. En effet, les cookies samesite=strict ne sont pas envoyés dans ce cas.

Stockage des données dans le navigateur

Nous pouvons contourner ça avec deux cookies : une pour la "reconnaissance générale", uniquement dans le but de dire : "Salut, John", et un autre pour les opérations de changements de données avec samesite=strict. Alors, une personne venant de l'extérieur du site verra un message de bienvenue, mais les paiements devront être initiés depuis le site de la banque, pour que le second cookie soit envoyé.

- samesite=lax

Une approche plus relax qui protège aussi des XSRF et qui n'entrave pas l'expérience utilisateur.

Le mode lax, tout comme strict, interdit au navigateur d'envoyer des cookies quand venu de l'extérieur du site, mais ajoute une exception.

Un cookie samesite=lax est envoyé lorsque deux conditions sont réunies :

- La méthode HTTP est "safe" (e.g. GET, mais pas POST).

  La liste complète des méthodes HTTP safes est dans la spécification RFC7231. Basiquement ce sont des méthodes qui peuvent être utilisées pour lire, mais pas pour écrire de données. Elles ne doivent pas effectuer d'opérations de modifications de données. Suivre un lien c'est toujours du GET, la méthode safe.

Stockage des données dans le navigateur

- L'opération effectue une navigation de haut niveau (change l'URL dans la barre d'adresse).

  C'est généralement vrai, mais si la navigation est effectuée dans une <iframe>, alors ce n'est pas du haut-niveau. Aussi, les méthodes JavaScript n'effectuent aucune navigation, alors elles ne conviennent pas.

  Donc, que fait samesite=lax, il permet les opérations basiques "va à l'URL" à avoir des cookies. E.g. ouvrir un lien depuis des notes satisfait ces conditions.

  Mais quelque chose de plus compliqué, comme une requête depuis un autre site ou une soumission de formulaire, perd les cookies.

Si cela vous convient, alors ajouter samesite=lax n'entravera probablement pas l'expérience utilisateur et ajoutera une protection.

Dans l'ensemble, samesite est une bonne option.

Il y a un inconvénient : samesite est ignoré (non supporté) par les très vieux navigateurs, datant de 2017 et avant.

Donc si nous comptions uniquement sur samesite pour fournir une protection, alors les anciens navigateurs seraient vulnérables.

Mais nous pouvons assurément utiliser samesite avec d'autres mesures de protections, comme les tokens xsrf, pour ajouter une couche de défense additionnelle et donc, dans le futur, quand les anciens navigateurs mourront, nous pourrons probablement nous passer des tokens xsrf.

Stockage des données dans le navigateur

Les fonctions du cookie

Ici un petit ensemble de fonctions qui fonctionnent avec les cookies, plus pratiques que des modifications manuelles de document.cookie.

Il existe beaucoup de librairies de cookie pour ça, celles là sont à but démonstratifs. Elles fonctionnent complétement cependant.

getCookie(name)

Le moyen le plus court d'accéder à un cookie est d'utiliser une expression régulière.

La fonction getCookie(name) retourne un cookie avec le nom donné :

```
// Retourne le cookie correspondant au nom donné, ou undefined si non trouvé
function getCookie(name) {
  let matches = document.cookie.match(new RegExp(
    "(?:^|; )" + name.replace(/([\.$?*|{}\(\)\[\]\\\/\+^])/g, '\\$1') + "=([^;]*)"
  ));
  return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

Stockage des données dans le navigateur

Ici new RegExp est généré dynamiquement, pour faire correspondre : name=<value>.

Veuillez noter qu'un cookie a une valeur encodée, donc getCookie utilise une fonction decodeURIComponent intégrée pour la décoder.

setCookie(name, value, options)

Définit le cookie name à la valeur valeur avec path=/ par défaut (peut être modifié pour ajouter d'autres valeurs par défaut) :

```
function setCookie(name, value, options = {}) {
  options = {
    path: '/',
    // Ajoute d'autres valeurs par défaut si nécessaire
    ...options
  };
  if (options.expires instanceof Date) {
    options.expires = options.expires.toUTCString();
  }
let updatedCookie=encodeURIComponent(name)+"="+encodeURIComponent(value);
  for (let optionKey in options) {
    updatedCookie += "; " + optionKey;
    let optionValue = options[optionKey];
    if (optionValue !== true) {
      updatedCookie += "=" + optionValue;
    }
  }
  document.cookie = updatedCookie;
}
// Exemple d'utilisation :
setCookie('user', 'John', {secure: true, 'max-age': 3600});
```

Stockage des données dans le navigateur

deleteCookie(name)

Pour supprimer un cookie, nous pouvons l'appeler avec une date d'expiration négative :

```
// Retourne le cookie correspondant au nom donné, ou undefined si non trouvé
function deleteCookie(name) {
  setCookie(name, "", {
    'max-age': -1
  })
}
```

Mettre à jour ou supprimer doivent utiliser le même path et domain

Veuillez noter : quand nous mettons à jour ou supprimons un cookie, nous devons utiliser exactement les mêmes options path et domain que lorsque nous l'avions défini

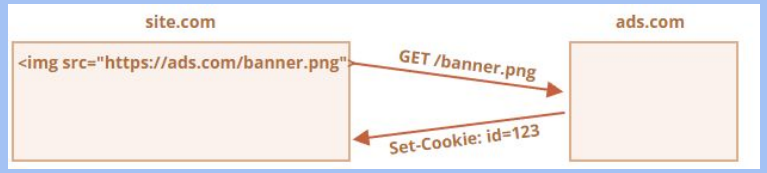
Stockage des données dans le navigateur

Les cookies tiers

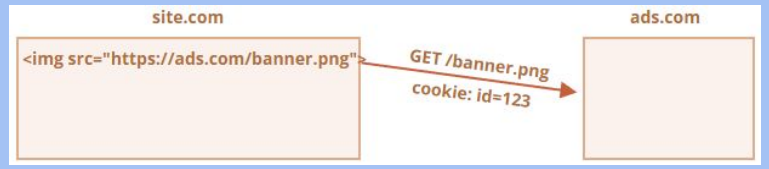
Un cookie est appellé "tiers" s'il est placé par un domaine autre que la page que l'utilisateur visite.

Par exemple :

1. Une page à site.com charge une bannière depuis un autre site : <img src="https://ads.com/banner.png">.
2. Le long de la bannière, le serveur distant à ads.com peut définir un entête Set-Cookie avec un cookie type id=1234. Un tel cookie provenant du domaine ads.com, et ne sera visible que par ads.com :

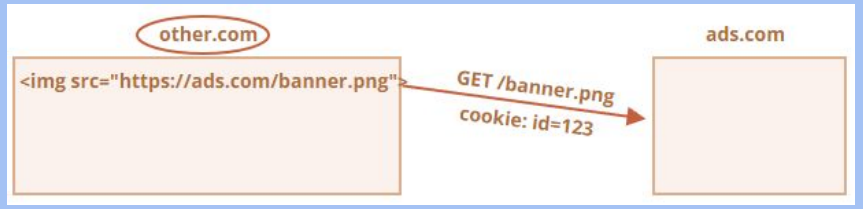


3. La prochaine fois que ads.com est accéder, le serveur distant récupère le cookie id et reconnait l'utilisateur :



4. Le plus important c'est que, quand un utilisateur bouge depuis site.com vers un autre site other.com, qui a lui aussi une bannière, alors ads.com récupère le cookie, comme elle appartient à ads.com, de fait il reconnait le visiteur et le tracke puisqu'il a bougé entre les sites :

Stockage des données dans le navigateur



Les cookies tiers sont traditionnellement utilisés pour le tracking et les services publicitaires, en raison de leur nature. Ils sont liés au domaine dont ils proviennent, donc ads.com peut tracker le même utilisateur entre différents sites, s'ils y accèdent tous.

Naturellement, certaines personnes n'aiment pas être trackées, donc les navigateurs permettent de désactiver ce genre de cookie.

Aussi, les navigateur modernes mettent en place des politiques spéciales pour de tels cookies :

- Safari ne permet pas du tout les cookies tiers
- Firefox vient avec une "black list" de domaines tiers où sont bloqués les cookies tiers.

Veuillez noter :

Si nous chargeons un script d'un domaine tiers, comme <script src="https://google-analytics.com/analytics.js">, et que ce script utilise document.cookie pour définir un cookie, alors un tel cookie n'est pas un cookie tiers.

Si un script définit un cookie, alors peu importe d'où vient le script – le cookie appartient au domaine de la page courante.

Stockage des données dans le navigateur

RGPD

Ce sujet n'est pas lié à JavaScript du tout, il s'agit de quelque chose à garder à l'esprit quand nous définissons des cookies.

Il y a une législation en Europe appellée **RGPD**, qui force les sites web à suivre un ensemble de règle pour respecter la vie privée de l'utilisateur. L'une de ces règles est de nécessiter une permission explicite de l'utilisateur pour les cookies de tracking.

Veuillez noter, ça concerne seulement les cookies de **tracking** / **identification** / **autorisation**.

Donc, si nous définissons un cookie pour sauvegarder certaines informations, mais sans tracker ni identifier l'utilisateur, nous sommes libre de le faire.

Mais si nous allons définir un cookie avec une session d'authentification ou un identifiant de tracking, alors l'utilisateur doit le permettre.

Les sites web ont généralement deux variantes pour suivre le RGPD. Vous devez les avoir déjà vu sur le web :

Si un site web veut définir des cookies de tracking uniquement pour les utilisateurs authentifiés.

Stockage des données dans le navigateur

Pour faire ça, le formulaire d'enregistrement doit avoir une case à cocher comme "*Accepter la politique sur la vie privée*" (qui décrit comment les cookies sont utilisés), l'utilisateur doit la cocher, et alors le site web est libre de définir des cookies d'authentification.

Si un site web veut définir des cookies de tracking pour tout le monde.

Pour faire ça légalement, un site web affiche une fenêtre contextuelle "de démarrage" pour les nouveaux venus, et nécessite qu'ils acceptent les cookies. Alors le site web peut les définir et laisser les gens voir le contenu.

Ça peut être dérangeant pour les nouveaux visiteurs cependant. Personne n'aime voir de telles fenêtres contextuelles "doit cliquer" plutôt que le contenu. Mais le RGPD requiert un accord explicite.

Le RGPD ne concerne pas uniquement les cookies, ça concerne aussi les trucs d'ordres personnels. Mais ça va au delà de notre portée.

Stockage des données dans le navigateur

LocalStorage, sessionStorage

Les objets de stockage Web localStorage et sessionStorage permettent d'enregistrer les paires clé/valeur dans le navigateur.

Ce qui est intéressant à leur sujet, c'est que les données survivent à une actualisation de la page (pour sessionStorage) et même à un redémarrage complet du navigateur (pour localStorage). Nous verrons cela très bientôt.

Nous avons déjà des cookies. Pourquoi des objets supplémentaires ?

Contrairement aux cookies, les objets de stockage Web ne sont pas envoyés au serveur à chaque requête. Grâce à cela, nous pouvons stocker beaucoup plus. La plupart des **navigateurs autorisent au moins 5 mégaoctets de données** (ou plus) et ont des paramètres pour configurer cela.

Contrairement aux cookies également, le serveur ne peut pas manipuler les objets de stockage via les en-têtes HTTP. Tout se fait en **JavaScript**.

Le stockage est lié à l'origine (triplet domaine/protocole/port). Autrement dit, différents protocoles ou sous-domaines impliquent différents objets de stockage, ils ne peuvent pas accéder aux données les uns des autres.

Stockage des données dans le navigateur

Les deux objets de stockage fournissent les mêmes méthodes et propriétés :

- **setItem**(key, value) – stocke la paire clé/valeur.
- **getItem**(key) – récupère la valeur par clé.
- **removeItem**(key) – supprime la clé avec sa valeur.
- **clear**() – supprime tout.
- **key**(index) – récupère la clé sur une position donnée.
- **length** – le nombre d'éléments stockés.

Comme vous pouvez le voir, c'est comme une collection Map (setItem/getItem/removeItem), mais permet également l'accès par index avec key(index).

Démo localStorage

Voyons voir comment ça fonctionne.

Les principales caractéristiques de localStorage sont les suivantes :

- Partagé entre tous les onglets et fenêtres d'une même origine.
- Les données n'expirent pas. Il reste après le redémarrage du navigateur et même le redémarrage du système d'exploitation.

Stockage des données dans le navigateur

Par exemple, si vous exécutez ce code...

```
localStorage.setItem('test', 1);
```

...Et fermez/ouvrez le navigateur ou ouvrez simplement la même page dans une autre fenêtre, alors vous pouvez l'obtenir comme ceci :

```
alert( localStorage.getItem('test', 1) );
```

Il suffit d'être sur la même origine (domaine/port/protocole), le chemin de l'url peut être différent.

Le localStorage est partagé entre toutes les fenêtres avec la même origine, donc si nous définissons les données dans une fenêtre, le changement devient visible dans une autre.

Boucle sur les clés

Comme nous l'avons vu, les méthodes fournissent la fonctionnalité "**get/set/remove** by key". Mais comment obtenir toutes les valeurs ou clés enregistrées ?

Malheureusement, les objets de stockage ne sont pas itérables.

Une façon consiste à boucler sur eux comme sur un tableau :

```
for (let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Stockage des données dans le navigateur

Chaînes uniquement

Veuillez noter que la clé et la valeur doivent être des chaînes. S'il s'agissait d'un autre type, comme un nombre ou un objet, il est automatiquement converti en chaîne de caractères.

sessionStorage

L'objet **sessionStorage** est beaucoup moins utilisé que **localStorage**. Les propriétés et les méthodes sont les mêmes, mais c'est beaucoup plus limité car sessionStorage n'existe que dans l'onglet actuel du navigateur. Un autre onglet avec la même page aura un stockage différent.

Mais il est partagé entre les iframes du même onglet (en supposant qu'ils proviennent de la même origine).

Les données survivent à l'actualisation de la page, mais pas à la fermeture/ouverture de l'onglet. Par exemple:

```
sessionStorage.setItem('test', 1);
```

...Puis actualisez la page. Maintenant, vous pouvez toujours obtenir les données :

```
alert( localStorage.getItem('test', 1) );
```

Stockage des données dans le navigateur

...Mais si vous ouvrez la même page dans un autre onglet et réessayez, le code ci-dessus renvoie null, ce qui signifie "rien trouvé".

C'est exactement parce que sessionStorage est lié non seulement à l'origine, mais également à l'onglet du navigateur. Pour cette raison, sessionStorage est utilisé avec parcimonie.

Événement de stockage

Lorsque les données sont mises à jour dans localStorage ou sessionStorage, l'événement storage se déclenche, avec les propriétés :

- **key** – la clé qui a été changée (null si .clear() est appelé).
- **oldValue** – l'ancienne valeur (null si la clé vient d'être ajoutée).
- **newValue** – la nouvelle valeur (null si la clé est supprimée).
- **url** – l'url du document où la mise à jour s'est produite.
- **storageArea** – objet localStorage ou sessionStorage où la mise à jour s'est produite.

L'important est que l'événement se déclenche sur tous les objets window où le stockage est accessible, sauf celui qui l'a provoqué.

Stockage des données dans le navigateur

Imaginez, vous avez deux fenêtres avec le même site dans chacune. Donc localStorage est partagé entre eux.

Vous pouvez ouvrir cette page dans deux fenêtres de navigateur pour tester le code ci-dessous.

Si les deux fenêtres écoutent **window.onstorage,** chacune réagira aux mises à jour qui se sont produites dans l'autre.

```
// se déclenche sur les mises à jour effectuées sur le même stockage à partir d'autres documents
window.onstorage = (event) => {
  // peut également utiliser window.addEventListener('storage', event => {
  if (event.key != "now") return;
  alert(event.key + ":" + event.newValue + " at " + event.url);
};
localStorage.setItem('now', Date.now());
```

Veuillez noter que l'événement contient également : **event.url** – l'url du document où les données ont été mises à jour.

De plus, **event.storageArea** contient l'objet de stockage – l'événement est le même pour **sessionStorage** et **localStorage**, donc event.storageArea fait référence à celui qui a été modifié. On peut même vouloir y remettre quelque chose, "répondre" à un changement. Cela permet à différentes fenêtres d'une même origine d'échanger des messages.

Les navigateurs modernes prennent également en charge Broadcast channel API, l'API spéciale pour la communication inter-fenêtre de même origine, elle est plus complète.