



DOM



- Document Object Model

API créé par Netscape en même temps que le JavaScript qui permet la manipulation du HTML en mémoire sous la forme d'un arbre

- W3C

Une norme existe depuis 1998, aujourd'hui dans sa 4e édition

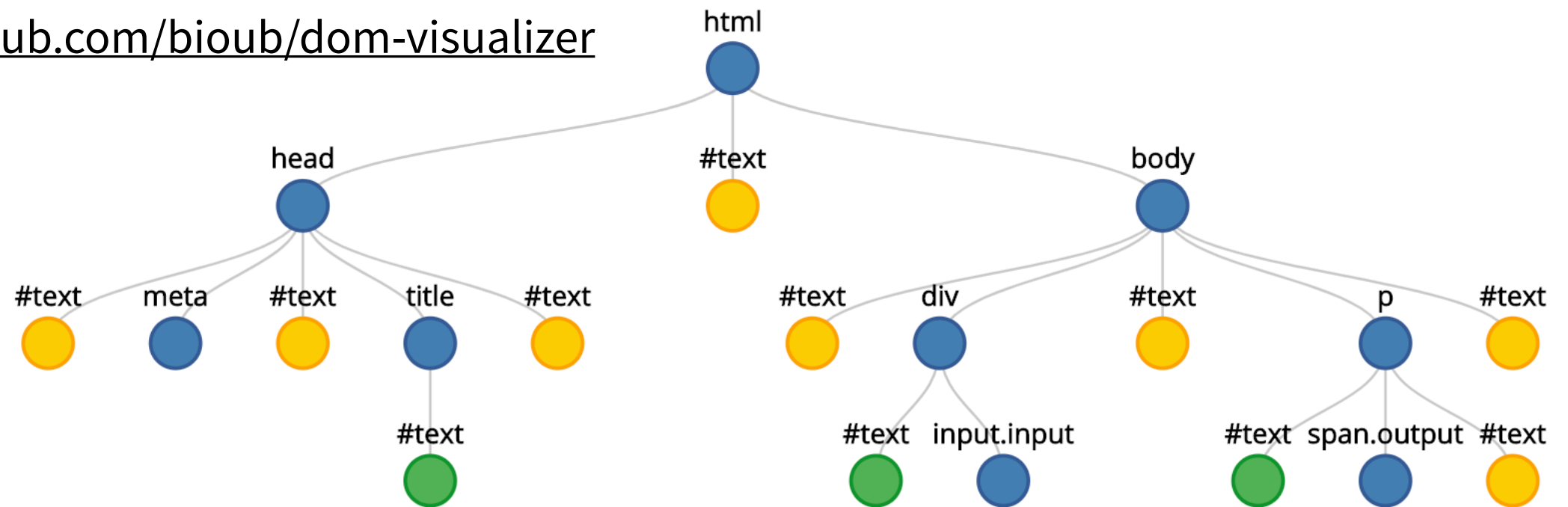
DOM Level 4 : <http://www.w3.org/TR/dom/>

DOM Level 3 Events : <http://www.w3.org/TR/DOM-Level-3-Events/>

HTML5 : <http://www.w3.org/TR/html5/>



- Représentation sous la forme d'un arbre  
<https://github.com/bioub/dom-visualizer>



```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DOM</title>
</head>
<body>
  <div>Name : <input class="input"></div>
  <p>Hello <span class="output"></span> !</div>
</body>
</html>
```



- Helloworld en DOM 1 (années 1990)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DOM</title>
  <script>
    function helloworld() {
      var inputElt = document.getElementsByTagName('input')[0];
      var spanElt = document.getElementsByTagName('span')[0];
      var text = document.createTextNode(inputElt.value);
      if (!spanElt.childNodes.length) {
        spanElt.appendChild(text);
      } else {
        spanElt.replaceChild(text, spanElt.firstChild);
      }
    }
  </script>
</head>
<body>
  <div>Name : <input id="input" onkeyup="helloworld()"></div>
  <p>Hello <span id="output"></span> !</div>
</body>
</html>
```



- Helloworld en DOM 1 (années 1990)
- Inconvénients :
  - écouter un événement sous forme d'attribut nécessite que la fonction helloworld soit globale
  - méthodes limitées pour retrouver un élément dans l'arbre
  - méthodes limitées pour créer du contenu (lourdeur de créer manuellement le noeud de texte)



- Helloworld en DOM 2 - 3 (années 2000)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DOM</title>
  <script>
    window.onload = function() {
      var inputElt = document.getElementById('input');
      var spanElt = document.getElementById('output');

      inputElt.onkeyup = function helloworld() {
        spanElt.textContent = this.value;
      };
    };
  </script>
</head>
<body>
  <div>Name : <input id="input"></div>
  <p>Hello <span id="output"></span> !</div>
</body>
</html>
```



- Helloworld en DOM 2 - 3 (années 2000)
- Inconvénients :
  - le code est en général en début de fichier, et s'exécute au chargement, il faut donc attendre que la page ait fini de charger
  - la méthode getElementById n'est disponible que sur l'objet document
  - la propriété onkeyup est unique, on risque un conflit si plusieurs callbacks écoutent le même événement sur le même élément (peut provenir d'une extension du navigateur)
  - on ne peut pas écouter l'événement dans une Event Phase spécifique



- Helloworld en DOM 4 (années 2010)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DOM</title>
</head>
<body>
  <div>Name : <input id="input"></div>
  <p>Hello <span id="output"></span> !</div>
  <script>
    (function() {
      'use strict';
      var inputElt = document.querySelector('#input');
      var spanElt = document.querySelector('#output');

      inputElt.addEventListener('input', function helloworld() {
        spanElt.innerText = this.value;
      });
    })();
  </script>
</body>
</html>
```





- Helloworld en DOM 4 (années 2010)
- Inconvénients :
  - module IIFE pour éviter les variables globales
  - mode strict pour désactiver des mauvais comportement
  - ambiguïté du mot clé this, l'objet dans lequel on est ? le champ input ?



- Helloworld en DOM 4 + ES2015 (années 2015+)

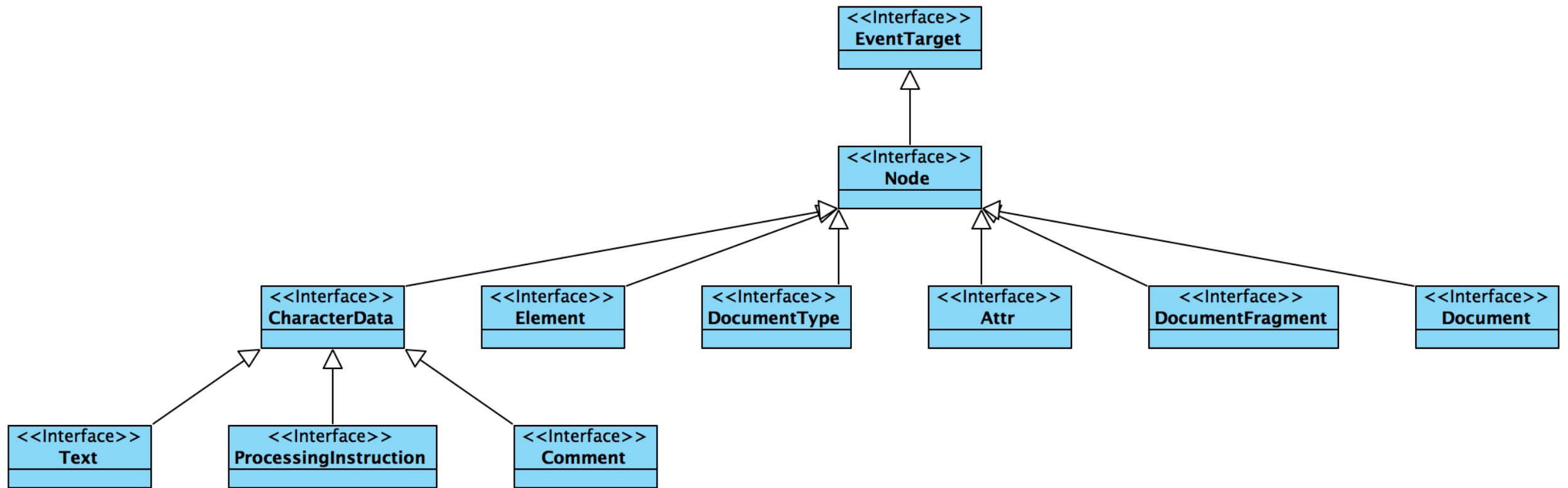
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DOM</title>
</head>
<body>
  <div>Name : <input id="input"></div>
  <p>Hello <span id="output"></span> !</div>
  <script type="module">
    const inputElt = document.querySelector('#input');
    const spanElt = document.querySelector('#output');

    inputElt.addEventListener('input', (event) => {
      spanElt.innerText = event.target.value;
    });
  </script>
</body>
</html>
```

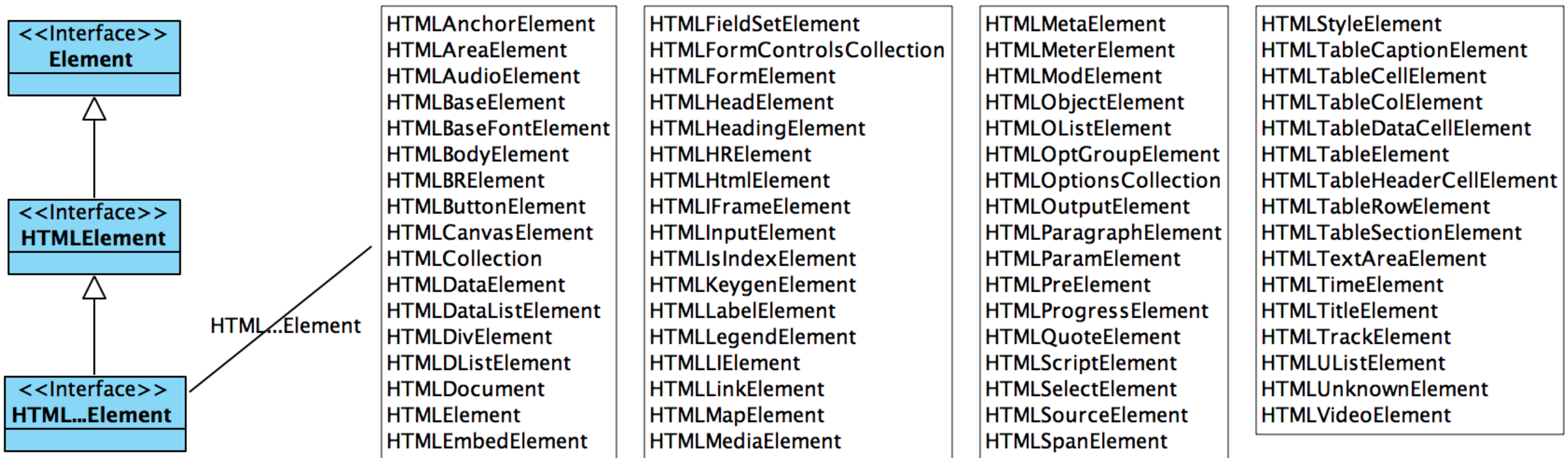


- Helloworld en DOM 4 (années 2010)
- Inconvénients :
  - les nouvelles syntaxes perdent la compatibilité avec les navigateurs anciens
  - `type="module"` également

# DOM - Interfaces




# DOM - HTML Interfaces





- Depuis n'importe quel noeud Element (DOM 4, IE9+) :
  - children : les noeuds Element enfants
  - firstElementChild : le premier noeud Element enfant
  - lastElementChild : le dernier noeud Element enfant
  - previousElementSibling : le frère Element précédent (même parent)
  - nextElementSibling : le frère Element suivant (même parent)
  - parentElement : le parent

## Browser compatibility

Desktop	Mobile				
Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support (on <a href="#">Element</a> )	1.0	<a href="#">3.5</a> (1.9.1)	9.0	10.0	4.0
Support on <a href="#">Document</a> and <a href="#">DocumentFragment</a> 	29.0	<a href="#">25.0</a> (25.0)	Not supported	16.0	Not supported



- Rechercher le premier noeud qui matche un sélecteur CSS
  - `document.querySelector('selecteur')`
  - `element.querySelector('selecteur')`
- Rechercher tous les éléments matchent un sélecteur CSS (retourne un NodeList, itérable)
  - `document.querySelectorAll('selecteur')`
  - `element.querySelectorAll('selecteur')`



- Tous les éléments : `document.all`
- Body : `document.body`
- Head : `document.head`
- Forms : `document.forms[ position ]` ou `document.forms[ id ]`
- Images : `document.images[ position ]` ou `document.images[ id ]`
- Scripts : `document.scripts[ position ]` ou `document.scripts[ id ]`
- Title : `document.title`





- Entre la balise ouvrante / fermante
  - `Element.textContent`
  - `Element.innerText`
  - `Element.innerHTML`
- Valeur d'un champs (input, select, textarea)
  - `Element.value`



- Sauf exception : les propriétés d'un Element porte le nom de l'attribut
- Ex : Element.id, HTMLElement.lang, HTMLElement.title, HTMLFormElement.action, HTMLFormElement.name...
- Exceptions :
  - Element.className (class est un mot clé de JS),
  - HTMLMetaElement n'a pas de propriété charset
  - ...
- Pour accéder à n'importe quel attribut :
  - Element.getAttribute( name )
  - Element.setAttribute( name, value )

# DOM - Attributs booléens



- Certains attributs ont un caractère booléen
- En HTML5 :
  - `<input type="text" required>`
- En XHTML
  - `<input type="text" required="required">`
- DOM :
  - `Element.required (true/false)`



- Pour stocker une valeur dans une balise dans un attribut "custom"
- DOM
  - `Element.dataset.monAttribut = "Valeur"`
- HTML
  - `<balise data-mon-attribut="valeur">`

# DOM - Ajouter des noeuds



- Pour créer un noeud on utilise une méthode commencer par *create*  
*document.createElement*, *document.createTextNode*, *document.createAttribute...*
- Pour l'insérer on utilise les méthodes *parent.insertBefore* ou *parent.appendChild*,  
*parent.replaceChild*, *parent.removeChild*
- De nouvelles méthodes sont en train de faire leur apparition (inspirée de jQuery) :  
*append*, *prepend*, *after*, *before*, *remove*, *replace*



- Les événements sont un mécanisme permettant d'exécuter du code au moment où une action se produit
- L'action peut être utilisateur (ex : clic) ou liée à un concept informatique (ex : la réponse d'une requête AJAX est reçue)
- Certains événements peuvent s'appliquer à tous les éléments (click, mousemove, keyup), d'autres sont spécifiques à certains éléments (submit d'un formulaire, timeupdate d'une vidéo...)
- Pour écouter des événements du DOM on utilise la méthode *addEventListener* :

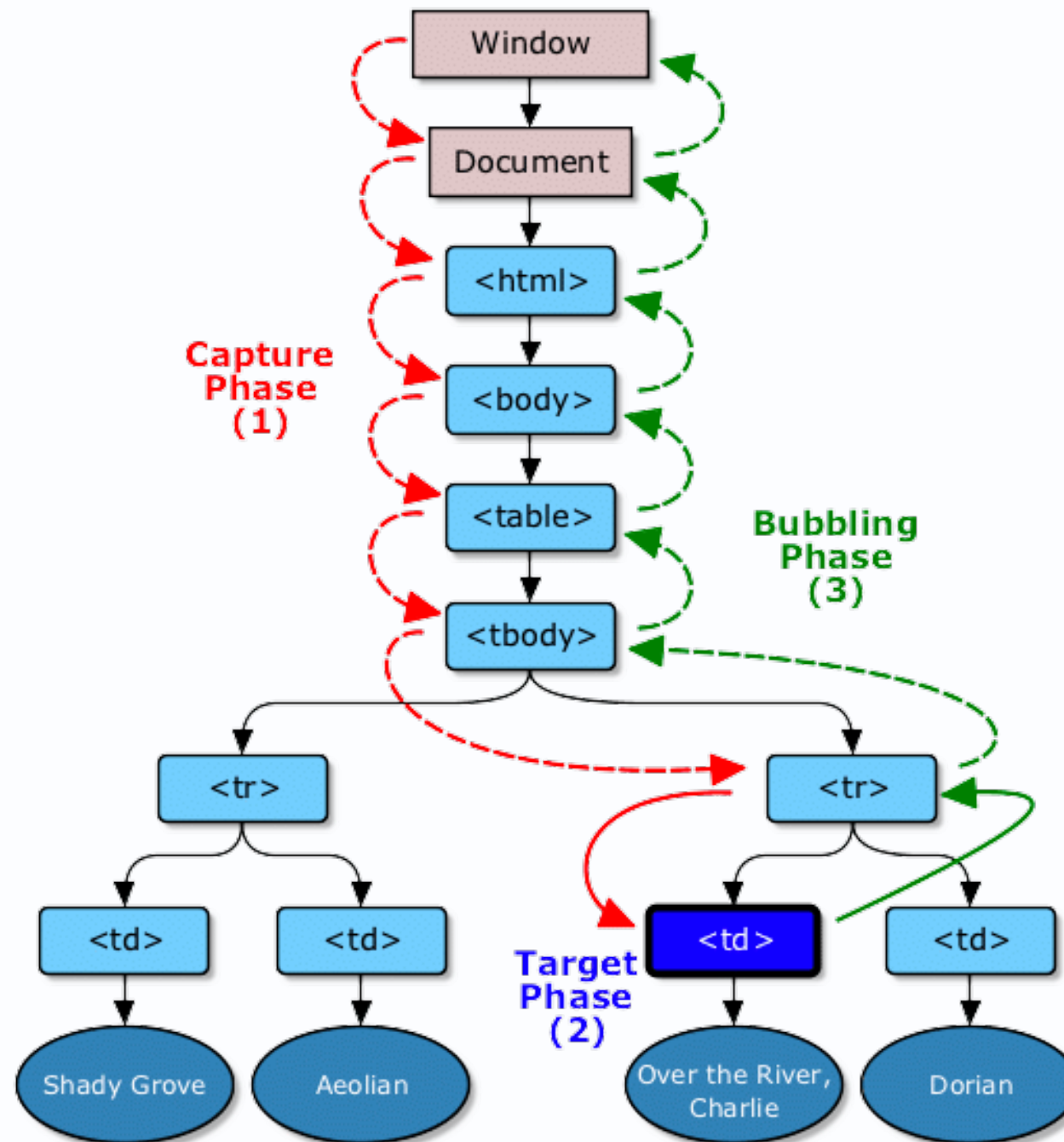
```
inputElt.addEventListener('input', (event) => {  
  spanElt.innerText = event.target.value;  
});
```



- Principaux événements souris :  
click, dblclick, mousedown, mousemove, mouseup, mouseenter, mouseleave, mouseover, mouseout, contextmenu
- Principaux événements clavier :  
keydown, keypress, keyup
- Principaux événements liés aux formulaires :  
submit, reset, focus, blur, input, beforeinput, change
- TouchEvents : événement tactiles (Smartphones, Tablettes...) :  
touchstart, touchend, touchmove
- PointerEvents : abstraction sur l'interface de pointeur (Souris, Tactile, Stylet...)  
[https://developer.mozilla.org/en-US/docs/Web/API/Pointer\\_events](https://developer.mozilla.org/en-US/docs/Web/API/Pointer_events)



- Pour les événements du DOM on peut distinguer différentes phases







- Pour écouter dans la phase de bubbling :

```
<body>
  <button class="btn">Clic moi</button>
  <div class="output"></div>
  <script type="module">
    const btnElt = document.querySelector('.btn');
    const outputElt = document.querySelector('.output');
    btnElt.addEventListener('click', () => {
      outputElt.textContent += 'button ';
    });
    document.body.addEventListener('click', () => {
      outputElt.textContent += 'body ';
    });
    // au clic du bouton : button body
  </script>
</body>
```



- Pour écouter dans la phase de capture :

```
<body>
  <button class="btn">Clic moi</button>
  <div class="output"></div>
  <script type="module">
    const btnElt = document.querySelector('.btn');
    const outputElt = document.querySelector('.output');
    btnElt.addEventListener('click', () => {
      outputElt.textContent += 'button ';
    });
    document.body.addEventListener('click', () => {
      outputElt.textContent += 'body ';
    }, true); // ou { useCapture: true }
    // au clic du bouton : body button
  </script>
</body>
```



- Pour écouter dans la phase de target :

```
<body>
  <button class="btn">Clic moi</button>
  <div class="output"></div>
  <script type="module">
    const outputElt = document.querySelector('.output');
    document.body.addEventListener('click', (event) => {
      if (event.target.classList.contains('btn')) {
        outputElt.textContent += 'button ';
      }
    });
    // au clic du bouton (uniquement) : button
  </script>
</body>
```

- Moins lisible mais permet d'écouter le clic du bouton y compris s'il n'existe pas encore



- Lorsqu'on écoute un événement du DOM, le callback est appelé avec un objet de type Event
- Il existent un certain nombre d'interface qui dérivent d'Event :  
<https://developer.mozilla.org/en-US/docs/Web/API/Event#Introduction>
- Exemple : MouseEvent, KeyboardEvent, PointerEvent...
- Un événement de type MouseEvent contiendra des propriétés telles que clientX, clientY (position de la souris sur la page)



- La méthode `preventDefault` permet d'empêcher l'action par défaut du navigateur lorsqu'il y en a une
- Exemple : submit d'un formulaire, click sur un lien, début de sélection de texte (`mousedown`)...

```
Number : <input class="number"/>
<script type="module">
  const inputElt = document.querySelector('.number');
  inputElt.addEventListener('beforeinput', (event) => {
    if (!event.data.match(/\d/)) {
      event.preventDefault();
    }
  });
</script>
```



- La méthode *stopPropagation* permet d'empêcher l'appel aux handlers du même événement sur les éléments ancêtres

```
<body>
  <button class="btn">Clic moi</button>
  <div class="output"></div>
  <script type="module">
    const btnElt = document.querySelector('.btn');
    const outputElt = document.querySelector('.output');
    btnElt.addEventListener('click', (event) => {
      event.stopPropagation();
      outputElt.textContent += 'button ';
    });
    btnElt.addEventListener('click', (event) => {
      outputElt.textContent += 'button ';
    });
    document.body.addEventListener('click', (event) => {
      outputElt.textContent += 'body ';
    });
    // au clic du bouton : button button
  </script>
</body>
```



- La méthode *stopImmediatePropagation* permet d'empêcher l'appel aux handlers du même événement sur le même élément

```
<body>
  <button class="btn">Clic moi</button>
  <div class="output"></div>
  <script type="module">
    const btnElt = document.querySelector('.btn');
    const outputElt = document.querySelector('.output');
    btnElt.addEventListener('click', (event) => {
      event.stopImmediatePropagation();
      outputElt.textContent += 'button ';
    });
    btnElt.addEventListener('click', (event) => {
      outputElt.textContent += 'button ';
    });
    document.body.addEventListener('click', (event) => {
      outputElt.textContent += 'body ';
    });
    // au clic du bouton : button
  </script>
</body>
```



# APIs Réseaux





- Microsoft permet dès 1999 dans IE5 la création de requête AJAX via une bibliothèque appelée XMLHttpRequest
- Permet à son application Outlook Web Access de récupérer des nouveaux emails sans devoir recharger toute la page
- Mozilla créer une interface appelée XMLHttpRequest compatible avec l'API de Microsoft
- Les requêtes peuvent être asynchrones ou synchrones (absurde car le navigateur serait indisponible le temps de la requête)
- L'API XMLHttpRequest a depuis reçu de nouvelles fonctionnalités en 2011 (requêtes cross-domain/cross-origin, progress events...)

# APIs Réseaux - XMLHttpRequest



- XMLHttpRequest est une fonction constructeur
- La méthode open permet de configurer la méthode HTTP et l'URL
- Des événements comme load, progress ou error peuvent être écoutés
- La méthode send permet d'envoyer la requête

```
// contact.json  
{ "firstName": "Bill", "lastName": "Gates" }
```

```
const xhr = new XMLHttpRequest();  
xhr.open('GET', 'contact.json');  
xhr.addEventListener('load', (event) => {  
  const contact = JSON.parse(xhr.response);  
  console.log(contact.firstName + ' ' + contact.lastName);  
  // Bill Gates  
});  
xhr.send();
```



- Fetch est un nouvel API plus moderne qu'XMLHttpRequest pour échanger avec le serveur
- L'API est basé sur les promesses
- Fetch n'est pas compatible avec tous les navigateurs :  
[https://developer.mozilla.org/fr/docs/Web/API/Fetch\\_API#Compatibilit%C3%A9\\_Navigateurs](https://developer.mozilla.org/fr/docs/Web/API/Fetch_API#Compatibilit%C3%A9_Navigateurs)
- Github a créé un Polyfill <https://github.com/github/fetch>
- Il existe une implémentation pour Node.js <https://github.com/bitinn/node-fetch>
- Une bibliothèque propose une version isomorphique (compatible Browser et Node.js) <https://github.com/matthew-andrews/isomorphic-fetch>
- Du fait d'être basé sur des promesses, certaines fonctionnalités ne sont pas possible (progress events)



- La méthode fetch retourne une promesse qui sera résolue en Response  
<https://developer.mozilla.org/en-US/docs/Web/API/Response>
- La Réponse est un stream qui sera lu en asynchrone via des méthodes comme json, blob ou text

```
fetch('contact.json')  
  .then((res) => res.json())  
  .then((contact) => {  
    console.log(contact.firstName + ' ' + contact.lastName);  
  });
```

# APIs Réseaux - Serveur HTTP Node.js



- Node.js permet la création d'un serveur HTTP rapidement en particulier avec une bibliothèque comme Express
- A installer avec npm :  
`npm install express`

```
const express = require('express');  
  
const app = express();  
  
app.get('/contact/1', (req, res) => {  
  res.json({ firstName: 'Bill', lastName: 'Gates' });  
});  
  
app.listen(3000, () => {  
  console.log('Server started');  
});
```

# APIs Réseaux - Sécuriser un API REST avec JWT



- JSON Web Token ou JWT est un système de token qui n'a pas besoin d'être stocké
- Il contient une signature qui sera vérifiée lors de futures requêtes
- Des bibliothèques permettent de manipuler les tokens JWT
- Côté Node.js (pour générer et vérifier) :  
<https://github.com/auth0/node-jsonwebtoken>
- Côté Navigateur (pour le décoder) :  
<https://github.com/auth0/jwt-decode>

# APIs Réseaux - Sécuriser un API REST avec JWT



## ▸ Générer un token JWT

```
app.post('/user/signin', express.json(), (req, res) => {  
  if (req.body.username !== user.username || req.body.password !==  
  user.password) {  
    return res.status(401).json({ error: 'Wrong credentials' });  
  }  
  const token = jwt.sign({ id: user.id, username: user.username }, secret);  
  res.json({ token });  
});
```

## ▸ Vérifier un token JWT

```
function jwtMiddleware(req, res, next) {  
  const token = req.headers.authorization;  
  try {  
    jwt.verify(token, secret);  
  } catch(err) {  
    return res.status(401).json({ error: 'Unauthorized' });  
  }  
  return next();  
}
```



- Les requêtes AJAX et Fetch permettent un échange via le protocole HTTP
- HTTP jusqu'à sa version 1.1 (la plus répandue actuellement) est à l'initiative du client (Client Pool)
- HTTP 2.0 permet la mise en place d'événement serveur (ou Server Push)
- Les WebSockets elles, permettent d'établir un canal de communication permanent entre le client et le serveur, leur permettant d'échanger dans les 2 sens





- Exemple avec le echo de websocket.org (retourne le message envoyé)

```
const ws = new WebSocket('wss://echo.websocket.org');
ws.addEventListener('open', () => {
  console.log('Connecté');
  ws.send('Bonjour');
  ws.send('Bye');
  setTimeout(() => {
    ws.close();
  }, 1000);
});
ws.addEventListener('close', () => {
  console.log('Déconnecté');
});
ws.addEventListener('message', (event) => {
  console.log('Message : ' + event.data);
});
// Connecté
// Message : Bonjour
// Message : Bye
// Déconnecté
```



- Côté Node.js plusieurs implémentations de serveur WebSocket existent
  - `ws`
  - `socket.io`
- Exemple : un serveur echo

```
const { Server } = require('ws');  
  
const server = new Server({ port: 8080 });  
  
server.on('connection', (ws) => {  
  ws.on('message', (message) => {  
    ws.send(message);  
  });  
});
```



- Si la communication ne se fait que dans le sens Serveur → Client on peut utiliser un nouvel API : EventSource

```
const es = new EventSource('http://localhost:3000/event-stream');
es.addEventListener('open', () => {
  console.log('Connecté');
});
es.addEventListener('message', (event) => {
  console.log(event.data);
});
es.addEventListener('close', () => {
  console.log('Déconnecté');
});
```



- Côté serveur, on va configurer une réponse HTTP qui ne se termine jamais

```
const express = require('express');

const app = express();

app.get('/event-stream', (req, res) => {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive',
    'Access-Control-Allow-Origin': '*',
  });
  res.write('\n');

  let id = 0;
  const timeout = setInterval(() => {
    res.write(`id: ${++id}\n`);
    res.write(`data: ${Date.now()}\n\n`);
  }, 1000);

  req.on('close', () => clearInterval(timeout));
});

app.listen(3000);
```



# APIs de Stockage

# APIs de Stockage - Introduction



- Historiquement, seuls les cookies permettent le stockage de données persistantes dans le navigateur
- Aujourd'hui :
  - Cookies
  - Local Storage
  - Session Storage
  - WebSQL (déprécié)
  - IndexedDB
  - FileSystem (non standard)
  - Cache

# APIs de Stockage - Introduction



- Synchrone vs Asynchrone

Certains API comme localStorage sont synchrone d'autres comme IndexedDB asynchrones

- Structure de données

On retrouve des APIs clé/valeurs et d'autres plus structurés

- Transactions

Certains API supporte les transactions (empêche l'écriture si une parmi un ensemble échoue)

- Comparaison

<https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/#comparison>

# APIs de Stockage - Same-origin policy



- L'origin est la combinaison de :
  - Schéma d'URI
  - Domaine
  - Eventuellement le port
- Exemples :  
<https://www.google.fr>  
<http://localhost:3000>
- Les API de Stockage impriment la same-origin policy, c'est à dire que les valeurs sont associées à une origin en particulier



# APIs de Stockage - Cookies



- Les cookies peuvent être créés par le serveur et envoyé automatiquement
- Côté on utilise la propriété `document.cookie` en lecture/écriture
- Il est conseillé d'utiliser une bibliothèque comme js-cookie  
<https://github.com/js-cookie/js-cookie>
- Exemple de valeur de `document.cookie` (Github) :  
`_ga=GA1.2.2043353612.1547462011; _octo=GH1.1.515337093.1547462012; tz=Europe%2FParis; has_recent_activity=1; _gat=1`
- Sans bibliothèque il faudrait écrire quelque chose comme :

```
const cookies = Array.from(document.cookie.matchAll(/(\w+)=([^\;]+)/g))
                        .map(([,key,value]) => ({key, value}));

// [
//   {key: "_ga", value: "1"},
//   {key: "_octo", value: "2"},
//   {key: "tz", value: "s"},
//   {key: "has_recent_activity", value: "1"},
//   {key: "_gat", value: "1"}
// ]
```

# APIs de Stockage - Local Storage et Session Storage



- Local Storage et Session Storage ont le même API
- Permettent comme les cookies la manipulation synchrone de données clés/valeurs
- Le localStorage est un espace de stockage associé au navigateur et qui persiste après sa fermeture
- Le sessionStorage est un espace de stockage associé à l'onglet d'un navigateur et est vidé une fois l'onglet fermé

```
1 <script>
```

```
2 localStorage.
```

- clear
- getItem
- key
- length
- removeItem
- setItem

```
1 <script>
```

```
2 sessionStorage.
```

- clear
- getItem
- key
- length
- removeItem
- setItem

# APIs de Stockage - Local Storage et Session Storage



- On utilise souvent le localStorage pour stocker le token JWT

```
const token = 'eyJhbGciOiJIUzI...';  
localStorage.setItem('token', token);  
  
console.log(localStorage.getItem('token')); // 'eyJhbGciOiJIUzI...'
```

# APIs de Stockage - IndexedDB



- Le localStorage est adapté au stockage de petites valeurs (tokens, locale, préférences, ...)
- Pour des valeurs plus conséquentes il faudrait privilégier IndexedDB qui :
  - est asynchrone
  - supporte les transactions
  - offre limite de stockage est bien plus grand, quelques Go contre quelques Mo (la limite dépend des navigateurs et de l'espace disque disponible)
- L'API est un peu complexe à utiliser, on pourrait privilégier des bibliothèques comme
  - idb <https://github.com/jakearchibald/idb>
  - localForage <https://localforage.github.io/localForage/>

# APIs de Stockage - IndexedDB



## ▸ Exemple

```
const openRequest = indexedDB.open('address-book', 1);

openRequest.addEventListener('upgradeneeded', () => {
  const db = openRequest.result;
  const store = db.createObjectStore('contacts', { autoIncrement: true });
  store.put({firstName: 'Bill', lastName: 'Gates'});
  store.put({firstName: 'Steve', lastName: 'Jobs'});
});

openRequest.addEventListener('success', (event) => {
  const db = openRequest.result;
  const request = db.transaction('contacts').objectStore('contacts').getAll();

  request.addEventListener('success', (event) => {
    console.log(request.result);
    // [
    //   { firstName: 'Bill', lastName: 'Gates' },
    //   { firstName: 'Steve', lastName: 'Jobs' }
    // ]
  });
});
```



# Workers

# Workers - Introduction



- Le navigateur exécute le code JavaScript dans 1 seul thread
- Ce thread fait également le rendu de la page
- Lorsqu'un traitement JavaScript devient trop lourd, bloquer le thread empêchera le navigateur de dessiner la page, rendant ainsi toute interaction impossible
- Pour éviter cela, on peut utiliser un Worker qui exécutera le code JavaScript lourd dans un thread séparé et communiquera avec le thread principal via des événements

# Workers - Exemple



- Exemple : recherche de nombre premiers  $> 2^{52}$

```
// prime-nbs.js
for (let nb = 2 ** 52; nb < Number.MAX_SAFE_INTEGER; nb++) {
  let isPrime = true;

  for (let i = 2; i < Math.sqrt(nb) + 1; i++) {
    if (nb % i === 0) {
      isPrime = false;
      break;
    }
  }

  if (isPrime) {
    postMessage(nb);
  }
}
```

```
const worker = new Worker('prime-nbs.js');
let date = Date.now();

worker.addEventListener('message', (event) => {
  console.log(event.data);
  console.log('Trouvé en : ' + ((Date.now() - date) / 1000) + 's');
  date = Date.now();
  // 4503599627370517
  // Trouvé en : 0.603s
  // ...
});
```