



# Formation ReactJS, programmation avancée

Romain Bohdanowicz

Twitter : @bioub - Github : <https://github.com/bioub>

<http://formation.tech/>



- Romain Bohdanowicz

Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle

- Expérience

Formateur/Développeur Freelance depuis 2006

Plus de 10 000 heures de formation animées

- Langages

Expert : HTML / CSS / JavaScript / PHP / Java

Notions : C / C++ / Objective-C / C# / Python / Bash / Batch

- Certifications

PHP 5 / PHP 5.3 / PHP 5.5 / Zend Framework 1

- Particularités

Premier site web à 12 ans (HTML/JS/PHP), Triathlète à mes heures perdues

- Et vous ?

Langages ? Expérience ? Utilité de cette formation ?



# ECMAScript 6

# ECMAScript 6 - Introduction



- ECMAScript 6, aussi connu sous le nom ECMAScript 2015 ou ES6 est la plus grosse évolution du langage depuis sa création (juin 2015)  
<http://www.ecma-international.org/ecma-262/6.0/>
- Le langage est enfin adapté à des application JS complexes (modules, promesses, portées de blocks...)
- Pour découvrir les nouveautés d'ECMAScript 2015 / ES6  
<http://es6-features.org/>

# ECMAScript 6 - Compatibilité



- Compatibilité (novembre 2016) :
  - Dernière version de Chrome/Opera, Edge, Firefox, Safari : ~ 90%
  - Node.js 6 et 7 : ~ 90% d'ES6
  - Internet Explorer 11 : ~ 10% d'ES6
- Pour connaître la compatibilité des moteurs JS :  
<http://kangax.github.io/compat-table/>
- Pour développer dès aujourd'hui en ES6 et exécuter le code sur des moteurs plus anciens on peut utiliser des :
  - Compilateurs ou transpileurs : Babel, Traceur, TypeScript... Transforment la syntaxe ES6 en ES5
  - Bibliothèques de polyfills : core-js, es6-shim, es7-shim... Recréent les méthodes manquante en JS

# ECMAScript 6 - Portées de bloc



## ▸ let

- On peut remplacer le mot-clé var, par let et obtenir ainsi une portée de bloc
- La portée de bloc ainsi créée peut devenir une closure

```
for (var globalI=0; globalI<3; globalI++) {}  
console.log(typeof globalI); // number
```

```
for (let i=0; i<3; i++) {}  
console.log(typeof i); // undefined
```

```
// In 1s : 0 1 2  
for (let i=0; i<3; i++) {  
  setTimeout(() => {  
    console.log(i);  
  }, 1000);  
}
```

# ECMAScript 6 - Constantes



## ▸ Constantes

- Il est désormais possible de créer des constantes
- Comme pour let, les variables déclarées via const ont une portée de bloc
- Bonne pratique, utiliser const ou bien let lorsque ce n'est pas possible (plus jamais var)

```
if (true) {  
  const PI = 3.14;  
}  
  
console.log(typeof PI); // undefined  
  
const hello = function() {};  
// SyntaxError: Identifier 'hello' has already been declared  
const hello = function() {};
```

# ECMAScript 6 - Template literal



## ▸ Template literal / Template string

- Permet de créer une chaîne de caractères à partir de variables ou d'expressions
- Permet de créer des chaînes de caractères multi-lignes
- Déclarée avec un backquote ` (rarement utilisé dans une chaîne)

```
const prenom = 'Romain';
console.log(`Bonjour ${prenom} !`);

// ES5
// console.log('Bonjour ' + prenom + ' !');

const html = `
<table class="table">
  <tr><td>${prenom.toUpperCase()}</td></tr>
</table>
`;
```



# ECMAScript 6 - Fonctions fléchées



## ▸ Arrow Functions

- Plus courtes à écrire : (params) => retour.
- Si un seul paramètre, les parenthèses des paramètres sont optionnelles.
- Si le retour est un objet, les parenthèses du retour sont obligatoires.

```
const sum = (a, b) => a + b;  
const hello = name => `Hello ${name}`;  
const getCoords = (x, y) => ({x: x, y: y});
```

```
// ES5  
// var sum = function (a, b) {  
//   return a + b;  
// };  
// var hello = function (name) {  
//   return 'Hello ' + name;  
// };  
// var getCoords = function (x, y) {  
//   return {  
//     x: x,  
//     y: y,  
//   };  
// };
```

# ECMAScript 6 - Fonctions fléchées



▸ Avec bloc d'instructions

- Si les fonctions nécessitent plusieurs lignes, on peut utiliser un bloc { }
- Le mot clé return devient alors obligatoire

```
const isWon = (nbGiven, nbToGuess) => {  
  if (nbGiven < nbToGuess) {  
    return 'Too low';  
  }  
  
  if (nbGiven > nbToGuess) {  
    return 'Too high';  
  }  
  
  return 'Won !';  
};
```

# ECMAScript 6 - Fonctions fléchées



## ▸ Bonnes pratiques

- Attention à ne pas utiliser les fonctions fléchées pour déclarer des méthodes !
- Utiliser les fonctions fléchées pour les callback ou les fonctions hors objets
- Utiliser les method properties pour les méthodes
- Utiliser class pour les fonctions constructeurs

```
const globalThis = this;

const contact = {
  firstName: 'Romain',
  method1: () => { // Mauvaise pratique
    console.log(this === globalThis); // true
  },
  method2() { // Bonne pratique
    console.log(this === contact); // true
  }
};

contact.method1();
contact.method2();
```



## ▸ Paramètres par défaut

- Les paramètres d'entrées peuvent maintenant recevoir une valeur par défaut

```
const sum = function(a, b, c = 0) {  
  return a + b + c;  
};
```

```
console.log(sum(1, 2, 3)); // 6  
console.log(sum(1, 2)); // 3
```

```
// ES5  
// var sum = function(a, b, c) {  
//   if (c === undefined) {  
//     c = 0;  
//   }  
//   return a + b + c;  
// };
```



## ▸ Paramètres restants

- Pour récupérer les valeurs non déclarées d'une fonction on peut utiliser le REST Params
- Remplace la variable arguments (qui n'existe pas dans une fonction fléchée)
- La variable créée est un tableau (contrairement à arguments)
- Bonne pratique : ne plus utiliser arguments

```
const sum = (a, b, ...others) => {  
  let result = a + b;  
  
  others.forEach(nb => result += nb);  
  
  return result;  
};  
console.log(sum(1, 2, 3, 4)); // 10  
  
const sumShort = (...n) => n.reduce((a, b) => a + b);  
console.log(sumShort(1, 2, 3, 4)); // 10
```

# ECMAScript 6 - Spread Operator



## ▸ Spread Operator

- Le Spread Operator permet de transformer un tableau en une liste de valeurs.

```
const sum = (a, b, c, d) => a + b + c + d;

const nbs = [2, 3, 4, 5];
console.log(sum(...nbs)); // 14
// ES5 :
// console.log(sum(nbs[0], nbs[1], nbs[2], nbs[3]));

const otherNbs = [1, ...nbs, 6];
console.log(otherNbs.join(', ')); // 1, 2, 3, 4, 5, 6
// ES5 :
// const otherNbs = [1, nbs[0], nbs[1], nbs[2], nbs[3], 6];

// Clone an array
const cloned = [...nbs];
```

# ECMAScript 6 - Shorthand property



- Shorthand property

- Lorsque l'on affecte une variable à une propriété (maVar: maVar), il suffit de déclarer la propriété

```
const x = 10;  
const y = 20;  
  
const coords = {  
  x,  
  y,  
};  
  
// ES5  
// const coords = {  
//   x: x,  
//   y: y,  
// };
```

# ECMAScript 6 - Method properties



- Method properties
  - Syntaxe simplifiée pour déclarer des méthodes

```
const maths = {  
  sum(a, b) {  
    return a + b;  
  }  
};  
  
console.log(maths.sum(1, 2)); // 3  
  
// ES5  
// const maths = {  
//   sum: function(a, b) {  
//     return a + b;  
//   }  
// };
```



# ECMAScript 6 - Computed Property Names



## ▸ Computed Property Names

Permet d'utiliser une expression en nom de propriété

```
let i = 0;

const users = {
  [`user${++i}`]: { firstName: 'Romain' },
  [`user${++i}`]: { firstName: 'Steven' },
};

console.log(users.user1); // { firstName: 'Romain' }
```

```
/* ES5
var i = 0;
var users = {};
users['user ' + (++i)] = { firstName: 'Romain' };
users['user ' + (++i)] = { firstName: 'Steven' };

console.log(users.user1); // { firstName: 'Romain' }
*/
```

# ECMAScript 6 - Array Destructuring



## ▸ Déstructurer un tableau

- Permet de déclarer des variables recevant directement une valeur d'un tableau

```
const [one, two, three] = [1, 2, 3];  
console.log(one); // 1  
console.log(two); // 2  
console.log(three); // 3  
  
// ES5  
// var tmp = [1, 2, 3];  
// var one = tmp[0];  
// var two = tmp[1];  
// var three = tmp[2];
```

# ECMAScript 6 - Array Destructuring



- Déstructurer un tableau
  - Il est possible de ne pas déclarer un variable pour chaque valeur
  - Il est possible d'utiliser une valeur par défaut
  - Il est possible d'utiliser le REST Params

```
const [one, , three = 3] = [1, 2];  
console.log(one); // 1  
console.log(three); // 3  
  
const [romain, ...others] = ['Romain', 'Jean', 'Eric'];  
console.log(romain); // Romain  
console.log(others.join(', ')); // Jean, Eric
```

# ECMAScript 6 - Object Destructuring



- Déstructurer un objet
  - Comme pour les tableaux il est possible de déclarer une variable recevant directement une propriété

```
const {x: varX, y: varY} = {x: 10, y: 20};  
console.log(varX); // 10  
console.log(varY); // 20
```

# ECMAScript 6 - Object Destructuring



- Déstructurer un objet
  - Il est possible de nommer sa variable comme la propriété et d'utiliser shorthand property
  - Il est possible d'utiliser une valeur par défaut

```
const {x: x , y , z = 30} = {x: 10, y: 20};  
console.log(x); // 10  
console.log(y); // 20  
console.log(z); // 30
```

# ECMAScript 6 - Mot clé class



- Simplifie la déclaration de fonction constructeur
- Les classes n'existent pas pour autant en JavaScript, ce n'est qu'une syntaxe simplifiée (sucre syntaxique)
- Le contenu d'une classe est en mode strict

```
class Person {  
  constructor(firstName) {  
    this.firstName = firstName;  
  }  
  hello() {  
    return `Hello my name is ${this.firstName}`;  
  }  
}  
  
const instructor = new Person('Romain');  
console.log(instructor.hello()); // Hello my name is Romain  
  
// ES5  
// var Person = function(firstName) {  
//   this.firstName = firstName;  
// };  
// Person.prototype.hello = function() {  
//   return 'Hello my name is ' + this.firstName;  
// };
```



- Héritage avec le mot clé class
  - Utilisation du mot clé `extends` pour l'héritage
  - Utilisation de `super` pour appeler la fonction constructeur parent et les accès aux méthodes parents si redéclarée dans la classe

```
class Instructor extends Person {  
  constructor(firstName, speciality) {  
    super(firstName);  
    this.speciality = speciality;  
  }  
  hello() {  
    return `${super.hello()}, my speciality is ${this.speciality}`;  
  }  
}  
  
const romain = new Instructor('Romain', 'JavaScript');  
console.log(romain.hello()); // Hello my name is Romain, my speciality is  
JavaScript
```



# Modules ECMAScript



# Modules ECMAScript - Introduction



- JavaScript à sa conception
  - Objectif : créer des interactions côté client, après chargement de la page
  - Exemples de l'époque :
    - Menu en rollover (image ou couleur de fond qui change au survol)
    - Validation de formulaire
- JavaScript aujourd'hui
  - Applications front-end, back-end, en ligne de commande, de bureau, mobiles...
  - Applications pouvant contenir plusieurs centaines de milliers de lignes de codes (Front-end de Facebook > 1 000 000 LOC)
  - Il faut faciliter le travail collaboratif, en plusieurs fichiers et en limitant les risques de conflit

# Modules ECMAScript - Introduction

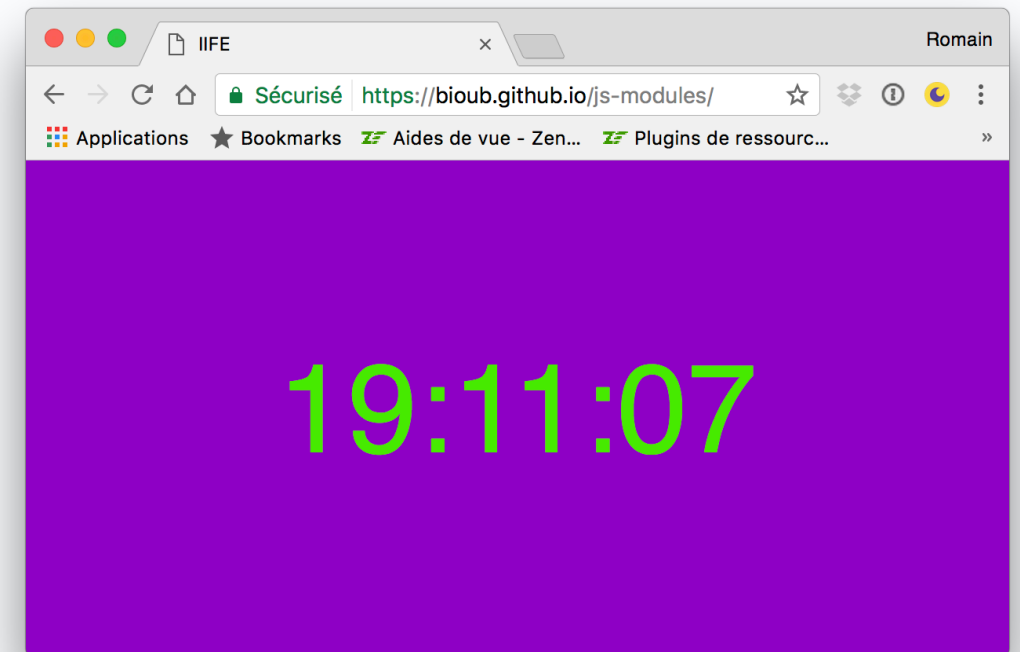
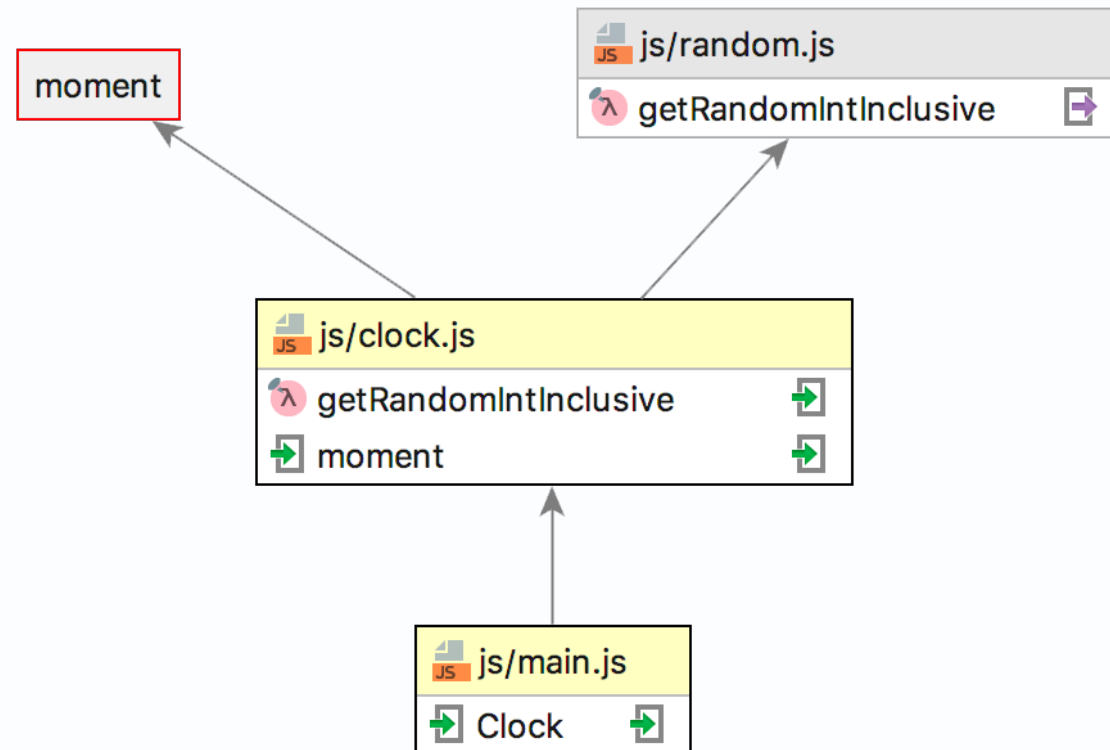


- Objectifs d'un module JavaScript
  - Créer une portée au niveau du fichier
  - Permettre l'export et l'import d'identifiants (variables, fonctions...) entre ces fichiers qui auront désormais leur propre portée
- Principaux systèmes existants
  - IIFE / Function Wrapper
  - CommonJS
  - AMD
  - UMD
  - SystemJS
  - ES6 (statiques mots clés import / export)
  - ESNext : import() (fonction asynchrone)

# Modules ECMAScript - Introduction



- Exemple utilisé pour la suite



- Le point d'entrée de l'application est le fichier `main.js`, qui dépend de `Clock` défini dans le fichier `clock.js`, qui dépend lui-même de `getRandomIntInclusive` du fichier `random.js` et `moment` défini dans le projet Open Source Moment.js
- Exemples : <https://github.com/bioub/js-modules/>
- Démo : <https://bioub.github.io/js-modules/>



- Portée de modules
  - Sans module la portée d'une fonction ou d'une variable déclarée dans un fichier serait globale.
  - Avec les modules une fonction sera locale au fichier.
- Import / Export
  - Une fonction ou un objet pouvant servir dans un autre fichier il faudra l'exporter.
  - Cela va créer l'API public du fichier (accessible de l'extérieur).
  - Un autre fichier devra importer les fonctions utilisées
- Mode Strict
  - Les modules ECMAScript sont par défaut en mode strict, il n'est donc pas nécessaire d'écrire *'use strict'*; en début de fichier.

# Module ECMAScript - Export



- Pour exporter une variable ou une fonction on utilise le mot clé export

```
export const getRandom = function() {  
  return Math.random();  
};  
  
export const getRandomArbitrary = function(min, max) {  
  return Math.random() * (max - min) + min;  
};  
  
export const getRandomInt = function(min, max) {  
  min = Math.ceil(min);  
  max = Math.floor(max);  
  return Math.floor(Math.random() * (max - min)) + min;  
};  
  
export const getRandomIntInclusive = function(min, max) {  
  min = Math.ceil(min);  
  max = Math.floor(max);  
  return Math.floor(Math.random() * (max - min + 1)) + min;  
};
```



- Il est également possible d'exporter en une seule fois en fin de fichier

```
const getRandom = function() {  
    return Math.random();  
};  
  
const getRandomArbitrary = function(min, max) {  
    return Math.random() * (max - min) + min;  
};  
  
const getRandomInt = function(min, max) {  
    min = Math.ceil(min);  
    max = Math.floor(max);  
    return Math.floor(Math.random() * (max - min)) + min;  
};  
  
const getRandomIntInclusive = function(min, max) {  
    min = Math.ceil(min);  
    max = Math.floor(max);  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
};  
  
export { getRandom, getRandomArbitrary, getRandomInt, getRandomIntInclusive };
```

# Module ECMAScript - Import



- Pour importer on utilise le mot clé `import`, associé à des accolades et le nom du fichier (l'extension est optionnelle)
- Lorsque que le fichier fait partie du projet, il est obligatoire de préfixer le fichier par `./` ou `../`
- Les modules ECMAScript ne peuvent être importée que statiquement en début de fichier. Pour des imports dynamiques il faut utiliser les modules CommonJS ou Dynamic Import (ESNext)

```
import { getRandomIntInclusive } from './random';

class Clock {
  // ...

  update() {
    let r = getRandomIntInclusive(0, 255);
    let g = getRandomIntInclusive(0, 255);
    let b = getRandomIntInclusive(0, 255);
    // ...
  }

  // ...
}
```

# Module ECMAScript - Tree Shaking



- Les imports étant statiques, des bundlers (bibliothèques de build) comme webpack ou Rollup peuvent analyser le code et éliminer du build les exports non importés
- Le build final ressemblera ainsi à :

```
const getRandomIntInclusive = function(min, max) {  
  min = Math.ceil(min);  
  max = Math.floor(max);  
  return Math.floor(Math.random() * (max - min + 1)) + min;  
};  
  
class Clock {  
  // ...  
  
  update() {  
    let r = getRandomIntInclusive(0, 255);  
    let g = getRandomIntInclusive(0, 255);  
    let b = getRandomIntInclusive(0, 255);  
    // ...  
  }  
  
  // ...  
}
```



# Module ECMAScript - Export/Import par défaut



- Il est possible de définir un export par défaut lorsqu'on a qu'une seule valeur à importer ou une valeur principale à importer
- Pour exporter on ajoute le mot clé *default*

```
export default class Clock {  
    // ...  
}
```

- Pour importer il faudra ne pas utiliser d'accolades

```
import Clock from './clock';  
  
let clockElt = document.querySelector('.clock');  
let clock = new Clock(clockElt);  
clock.start();
```

- Certains développeurs conseillent d'éviter les exports par défaut :  
<https://basarat.gitbooks.io/typescript/docs/tips/defaultIsBad.html>

# Module ECMAScript - Imports avancés



- On peut renommer un import, par exemple dans le cas où 2 identifiants auraient le même nom :

```
import { render as renderDom } from 'react-dom';  
import { App } from './App';  
  
renderDom(<App />, document.getElementById('root'));
```

- On peut également importer tous les exports dans un objet :

```
// serviceWorker.js  
export function register(config) {  
  // ...  
}  
  
export function unregister() {  
  // ...  
}
```

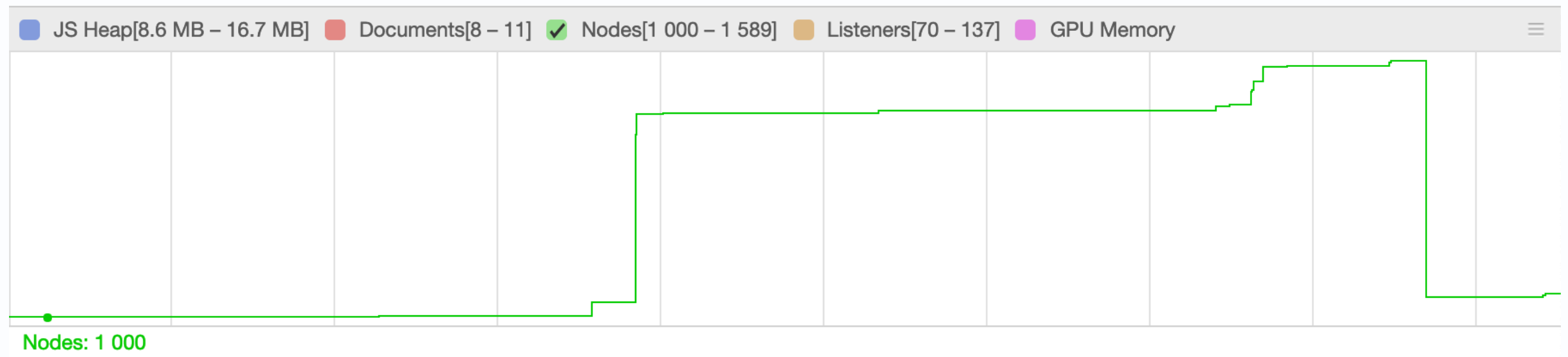
```
import * as serviceWorker from './serviceWorker';  
  
serviceWorker.unregister();
```



React



- Le DOM ou Document Objet Model est l'API du navigateur créé par Netscape en 1995 qui permet de manipuler le contenu de la page web
- Cet API est ancien même s'il reçoit des évolutions régulièrement
- Il est également très lourd, par exemple la page d'accueil de [formation.tech](https://formation.tech) va créer jusqu'à 1589 objet associés au DOM en mémoire



- Lorsqu'un composant React doit se rafraîchir (en appelant sa méthode render), il serait très coûteux de recréer tout les éléments du DOM qu'il contient. Pour éviter cela React met en place un "Virtual DOM"



- Voici un exemple de mini-framework sans Virtual DOM

```
class Component {
  _refresh() {
    this.host.innerHTML = '';
    this.render().forEach(elt => this.host.appendChild(elt));
  }
  setState(newState) {
    Object.assign(this.state, newState);
    this._refresh();
  }
}

function domRender(component, host) {
  component.host = host;
  component._refresh();
}
```

- Comme dans React, appeler la méthode `setState` ou `domRender` provoquera le rafraîchissement du composant en appelant sa méthode `render`.



- Comme React, le composant possède une méthode render qui construit le DOM

```
class ButtonCount extends Component {
  state = { count: 0 };
  increment = () => {
    this.setState({count: this.state.count + 1});
  };
  render() {
    const p = document.createElement('p');
    p.innerText = 'Démo : ';

    const button = document.createElement('button');
    button.innerText = this.state.count;
    button.onclick = this.increment;

    return [p, button];
  }
}

domRender(new ButtonCount(), document.querySelector('hello-component'));
```

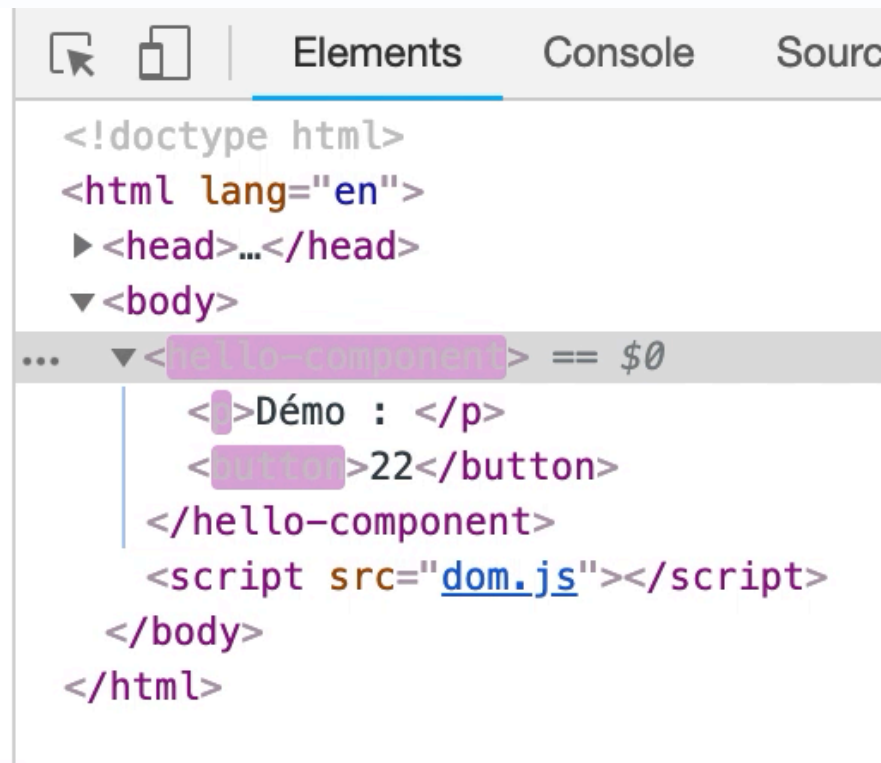
- On remarque que l'API DOM est lourd, si on pouvait chainer comme jQuery il n'y aurait que 2 lignes dans render
- Puis on peut demander le rendu dans une balise existante ici hello-component



- Lorsqu'on observe le résultat avec les DevTools de Chrome, on voit que l'ensemble du DOM associé au composant est rafraîchi

Démo :

22



```
<!doctype html>
<html lang="en">
  <head>...</head>
  <body>
    ... <hello-component> == $0
      <p>Démo : </p>
      <button>22</button>
    </hello-component>
    <script src="dom.js"></script>
  </body>
</html>
```



- Avec `React.createElement` on va construire avec un API plus moderne un arbre léger en mémoire appelé Virtual DOM

```
class ButtonCount extends React.Component {
  state = { count: 0 };
  increment = () => {
    this.setState({count: this.state.count + 1});
  };
  render() {
    return [
      React.createElement('p', null, 'Démo : '),
      React.createElement('button', {onClick: this.increment}, this.state.count),
    ];
  }
}

ReactDOM.render(
  React.createElement(ButtonCount),
  document.querySelector('hello-component'),
);
```





- ▶ Avec React et son Virtual DOM on remarque que le navigateur ne rafraîchit pas plus d'élément que nécessaire :

Démo :



```
Elements Console Source
<!doctype html>
<html lang="en">
  > <head>...</head>
  ▼ <body>
    ... ▼ <hello-component> == $0
      <p>Démo : </p>
      <button>22</button>
    </hello-component>
    <script src="../node_modules/react
    <script src="../node_modules/react
    <script src="react.js"></script>
  </body>
```



- L'exemple précédent est encore trop verbeux. Afin de le simplifier, React a créé une syntaxe appelée JSX pour construire le Virtual DOM d'un composant
- Le navigateur ne reconnaissant pas cette syntaxe on va utiliser un compilateur (en général Babel et son plugin `@babel/plugin-transform-react-jsx`) pour transformer le JSX en `React.createElement`

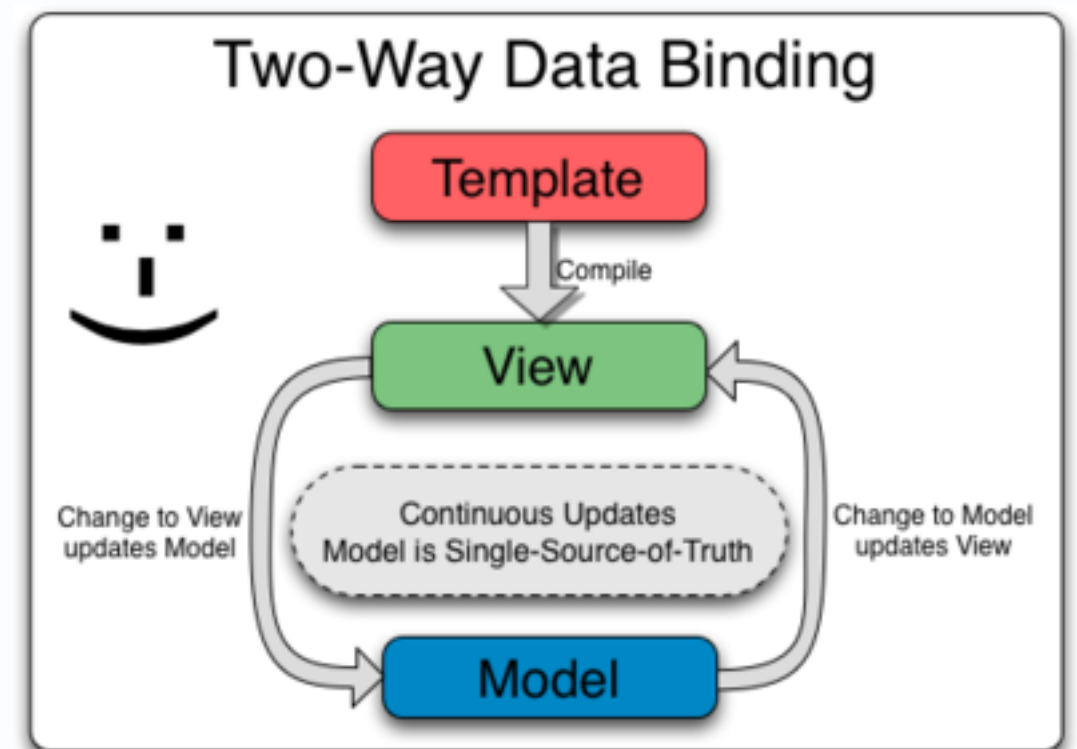
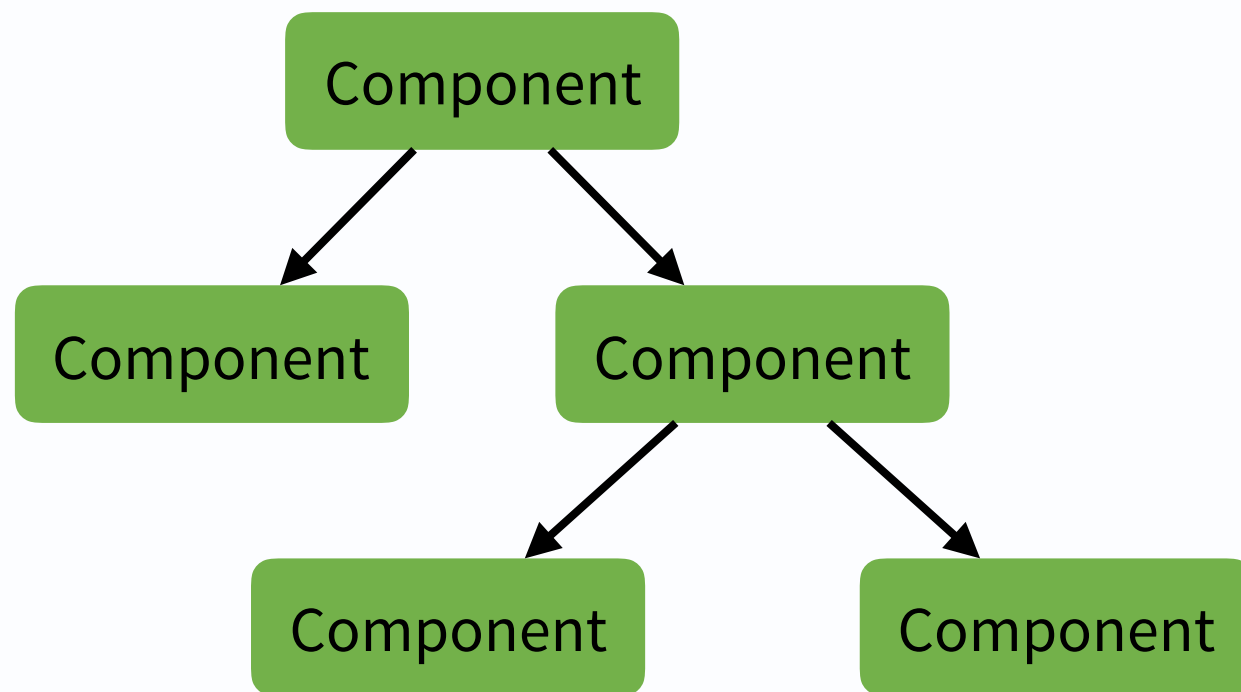
```
class ButtonCount extends React.Component {
  state = { count: 0 };
  increment = () => {
    this.setState({count: this.state.count + 1});
  };
  render() {
    return [
      <p>Démo :</p>,
      <button onClick={this.increment}>{this.state.count}</button>,
    ];
  }
}

ReactDOM.render(
  <ButtonCount />,
  document.querySelector('hello-component'),
);
```

# React - One Way Data Flow



- Par opposition aux frameworks de génération précédente comme AngularJS, Knockout ou Ember.js, les données circulent toujours dans un sens dans React : d'un composant parent vers un composant enfant. On parle de One-Way Data Flow, One-Way Data Binding ou Unidirectional Data Flow



# React - Outils de debug

