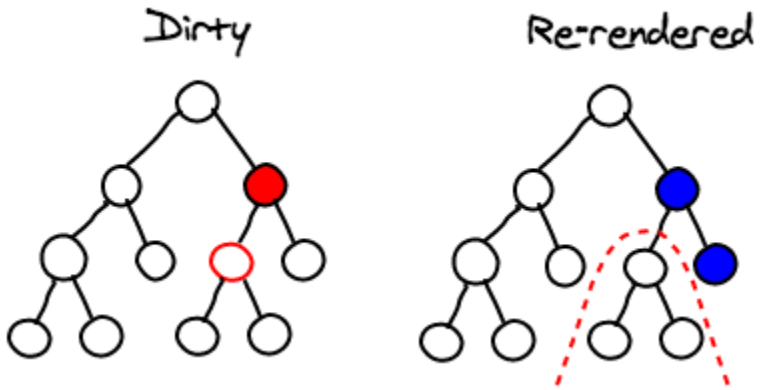
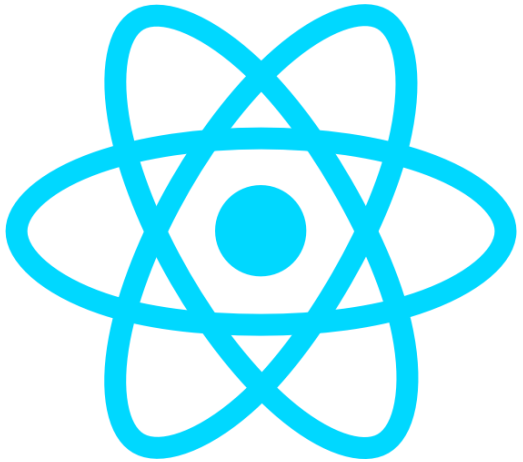


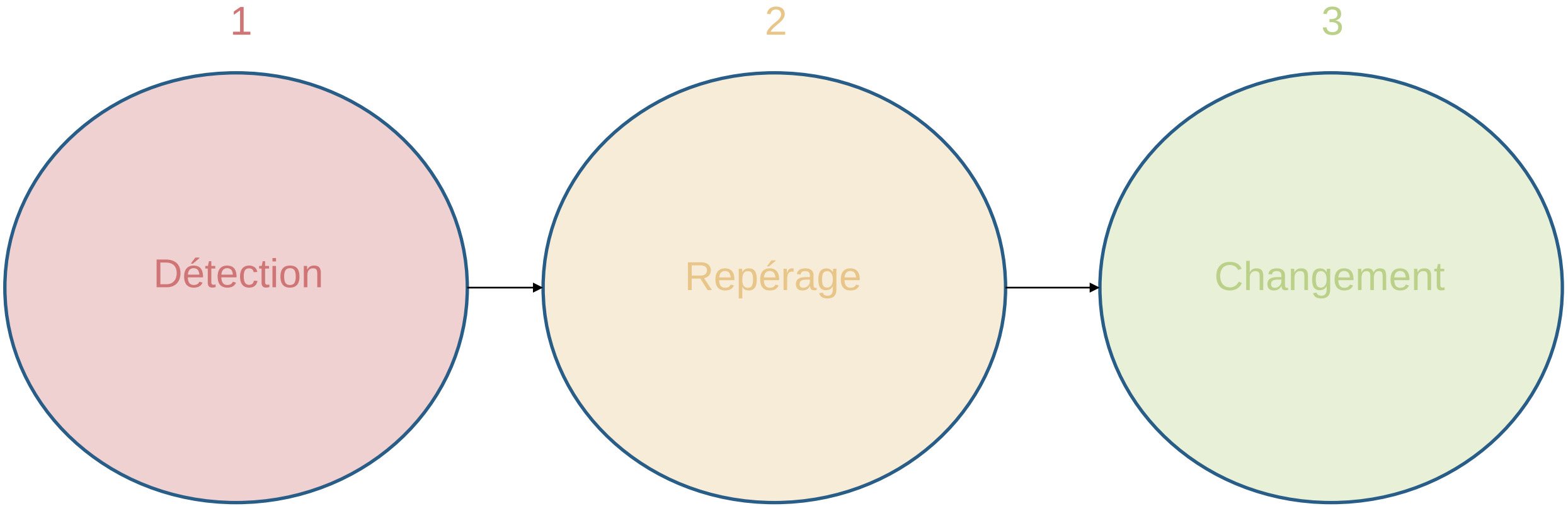
Performances

Animé par Mazen Gharbi


ember


2.0



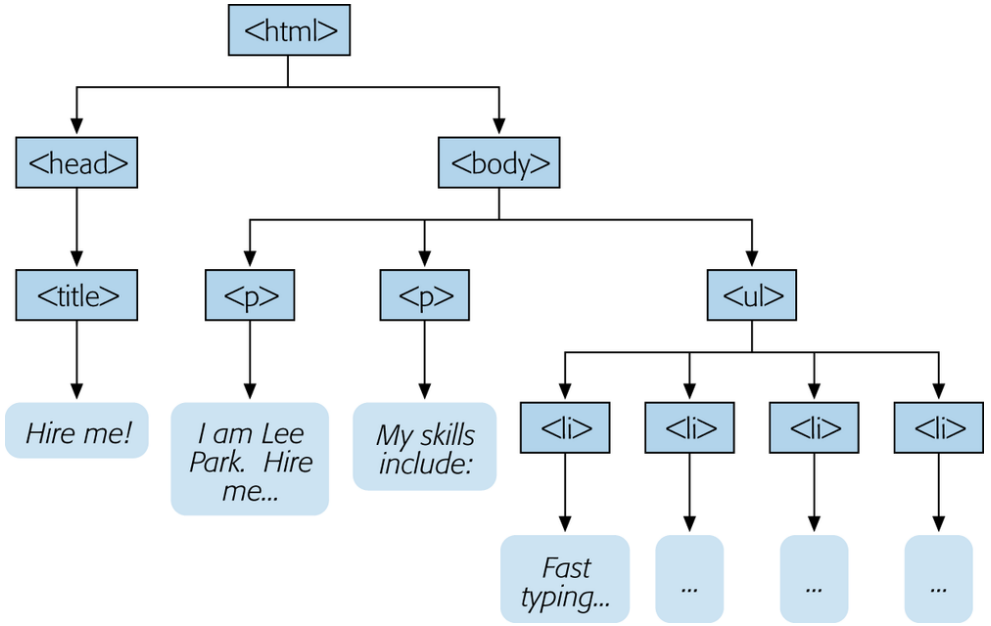

1
2
3
Détection
Repérage
Changement

Document Object Model

▷ Chaque page Web est représentée en interne en tant qu'arbre d'objets (DOM)

▷ Les opérations de réaffichage du DOM sont extrêmement coûteuses .

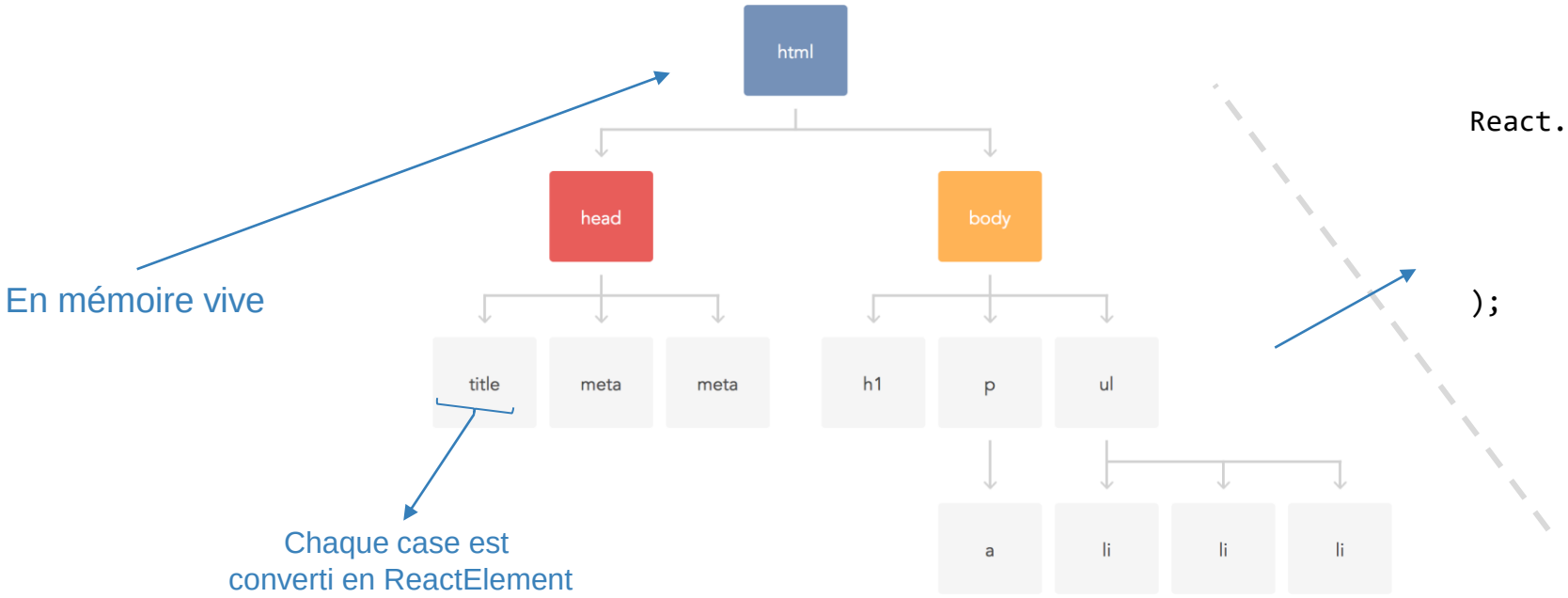
▷ React va donc chercher à minimiser le nombre de fois ou l'on va réafficher la page



Virtual DOM

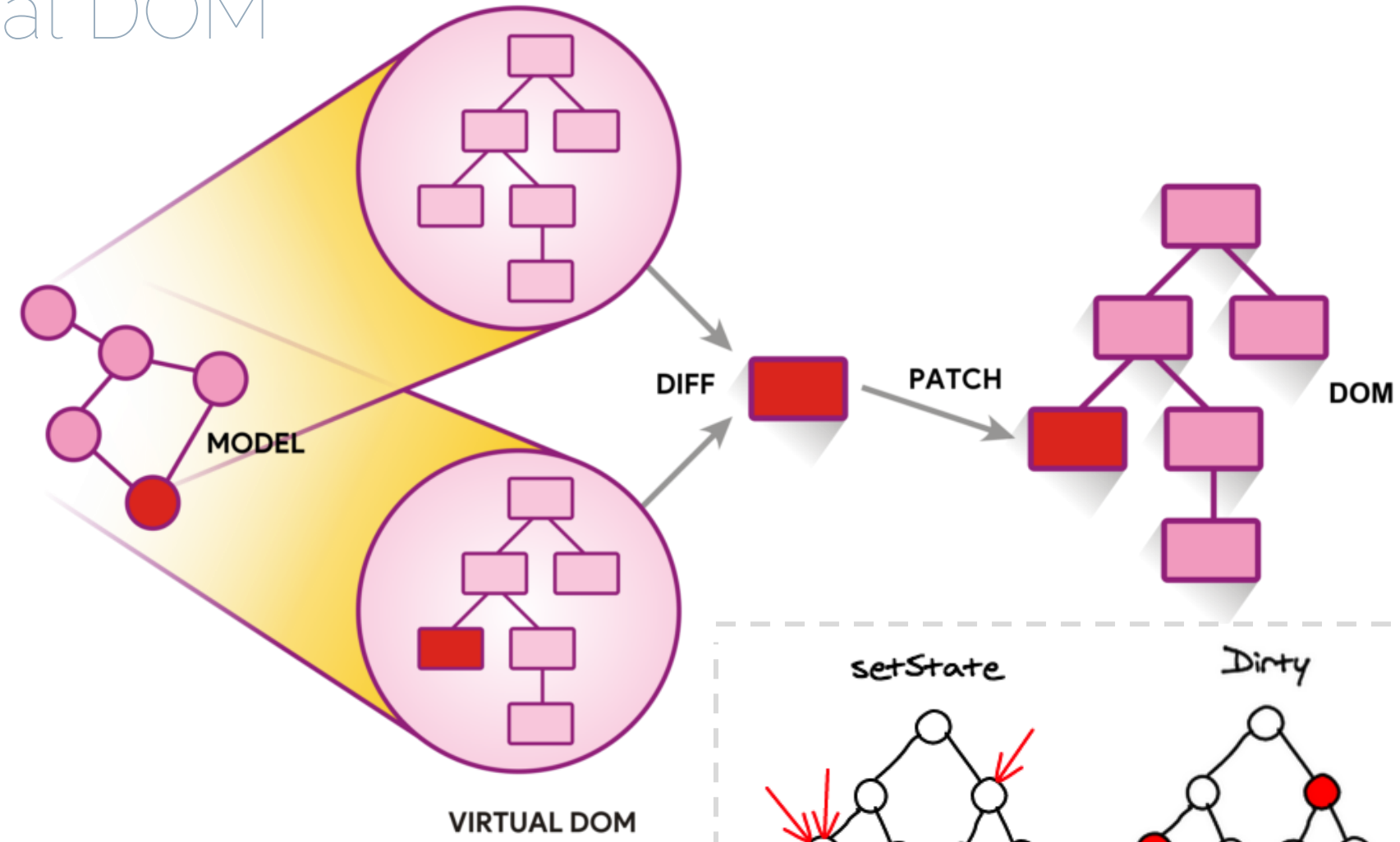
- React construit une représentation virtuelle du DOM actuel ;
  - *Ne pas confondre avec le Shadow DOM*

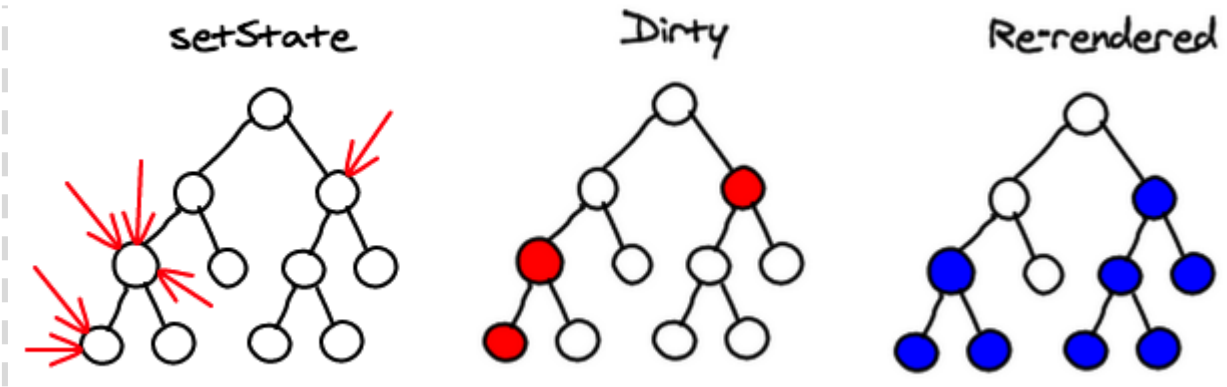
- Chaque nœud de l'arbre est représenté par un objet Javascript ;




```
React.createElement(
            "html",
            null,
            React.createElement("head", ...),
            React.createElement("body", ...)
);
```

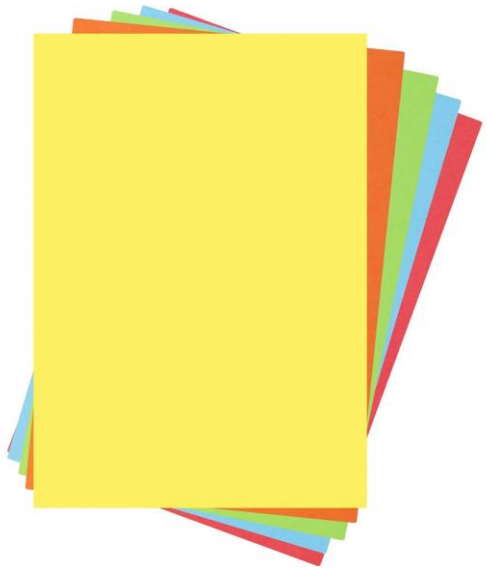
Virtual DOM


MODEL
DIFF
PATCH
DOM
VIRTUAL DOM


setState
Dirty
Re-rendered

Diffing

▷ Une fois le DOM virtuel mis à jour, React compare le DOM virtuel avec un snapshot du DOM virtuel prit juste avant la mise à jour ;



▷ En comparant le nouveau avec une version précédente, React détermine exactement quels objets DOM virtuels ont changés. Ce processus est appelé « diffing » ;

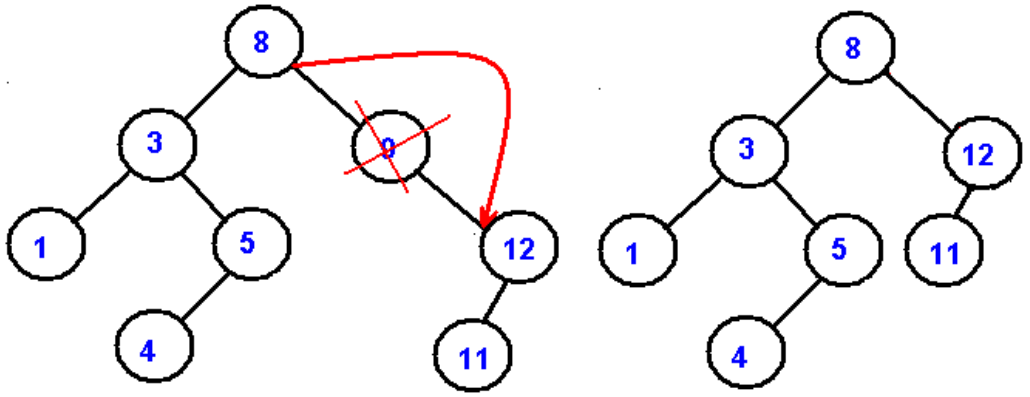
Diff

- Remplacer un arbre par un autre coûte cher en terme de performance
  - O(n^3)

- React évite à tout prix de modifier l'entièreté de l'arbre à chaque fois

- Il se base sur 2 conditions pour établir si une partie de l'arbre doit changer :
  1. 2 composants de la même classe généreront des arbres similaires et deux composants de classes différentes généreront des arbres différents ;
  2. Il est possible de fournir une clé unique pour les éléments stables sur différents rendus.

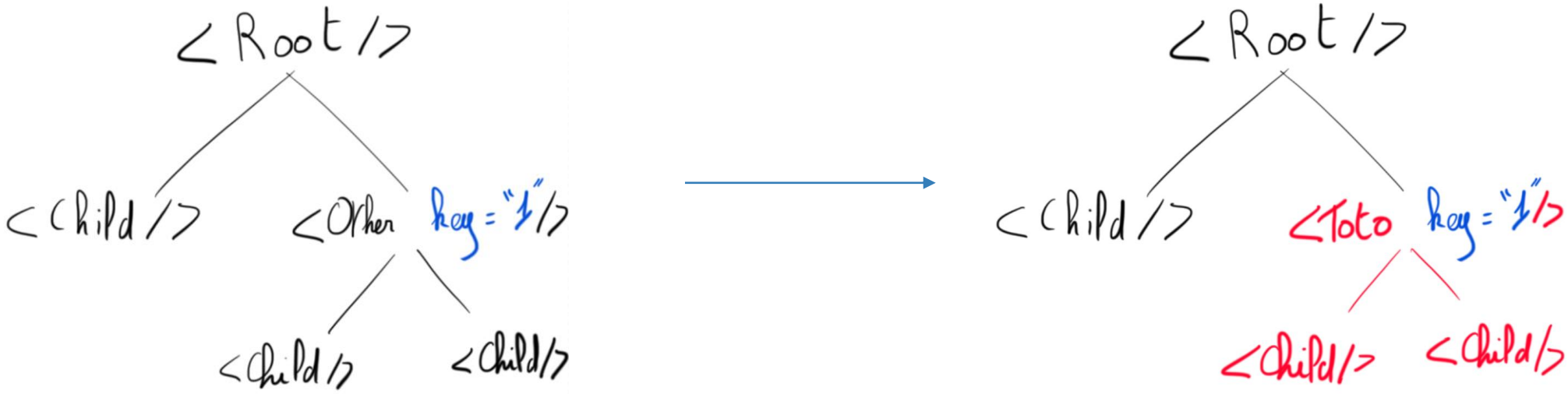
O(n^3) => O(n)



Diff – Différence de types

- Pour résumé :

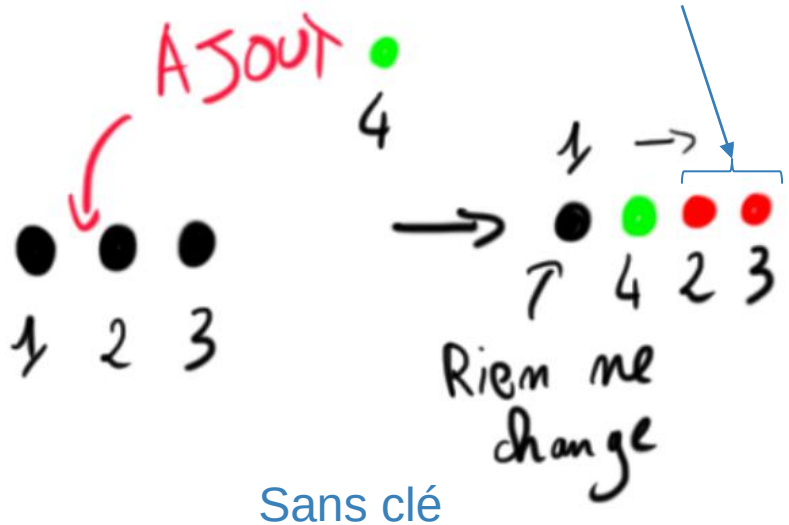
- Un sous-arbre doit être réaffiché si les types ne sont pas les mêmes



Diff – Différence de clés

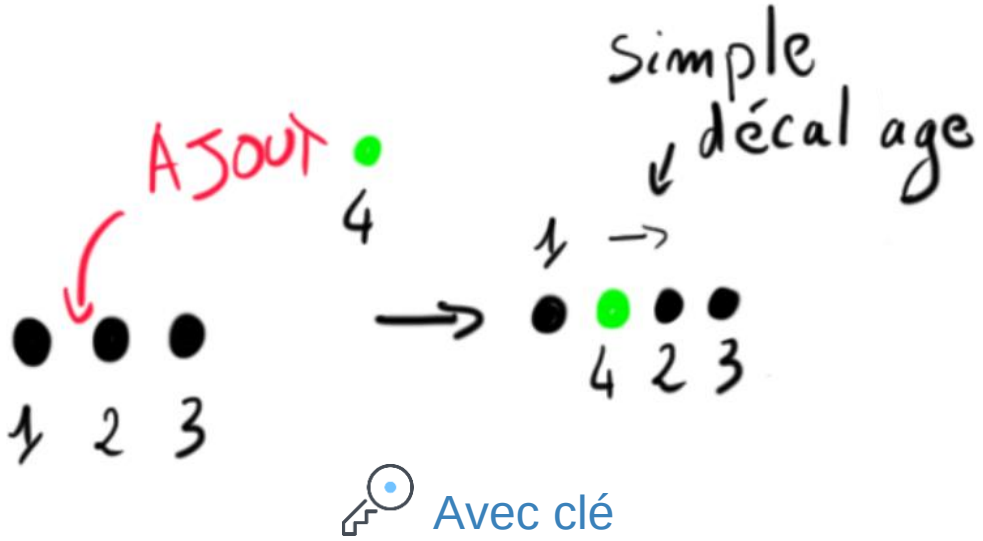
▷ Le réaffichage se fait pour l'élément dont la clé a changé uniquement

Ces 2 éléments ont été réaffichés



Sans clé

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```



Avec clé

```
<ul>
  <li key="uniq1">1</li>
  <li key="uniq2">2</li>
  <li key="uniq3">3</li>
</ul>
```

Bonnes pratiques

- Parfois, 2 composants différents peuvent générer une sortie similaire
  - React préconise dans ce cas de factoriser en 1 composant unique pour éviter le réaffichage suite au diffing ;

- Pensez à toujours appliquer une clé à vos éléments itérés ;

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map(number => (
        <li key={number.toString()}>{number}</li>
));
```

- *N'utilisez pas l'index de l'élément comme clé !*

Optimises les performances

▷ A chaque fois qu'une clé change, React considère que l'élément doit être réaffiché

▷ Or, il peut arriver qu'une propriété d'un composant change sans que la vue ne soit impactée

```
class MonComposant extends React.Component {
        // Nous comparons nous même l'état des states,
        // On renvoie true si on considère que la vue doit se mettre à jour
        // false sinon
        shouldComponentUpdate(nextProps, nextState) {
                if (nextProps === 'toto') {
                        return false;
                }

                return true;
        }
}
```

Mesurer les performances

▷ Afin d'améliorer les performances de notre application, il va être nécessaire de les mesurer !

▷ Pour ce faire, React propose une librairie : react-addons-perf

```
> npm install --save react-addons-perf
```

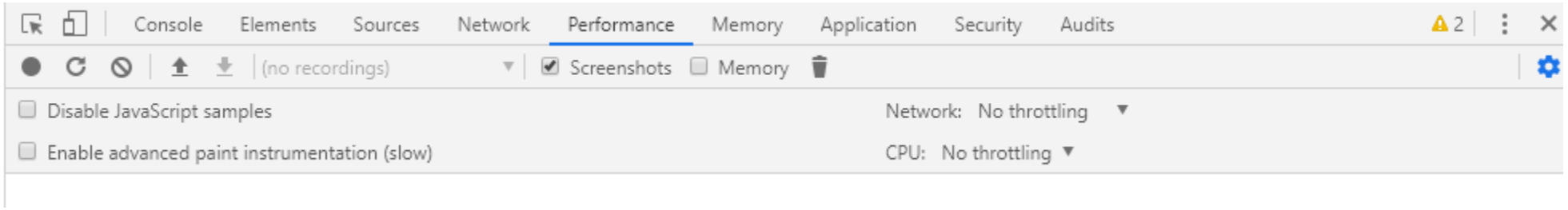
▷ Mais obsolète depuis React 16 !!

▷ Désormais, l'équipe React préconise le profiler du navigateur

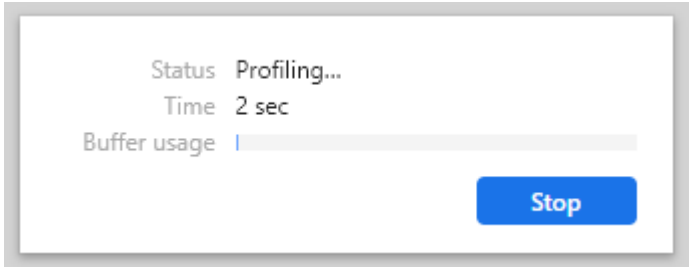
Mesurer les performances

▷ Ces manipulations doivent être faites avec toutes les extensions chromes désactivées ! (ou en navigation privée)

▷ Ouvrez votre console, et rendez vous sur l'onglet « Performance »



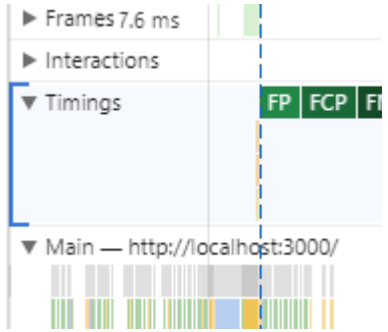
▷ Lancer le « record » puis rechargez la page



Mesurer les performances

Plus d'infos

- Tout va se passer dans la vue « Timings »



- Et voici le résultat :

Self Time	Total Time	Activity
5.9 ms 31.4 %	12.0 ms 64.4 %	App [mount]
5.8 ms 30.9 %	5.8 ms 30.9 %	Test [mount]
3.8 ms 20.2 %	15.8 ms 84.6 %	(React Tree Reconciliation: Completed Root)
1.2 ms 6.2 %	1.2 ms 6.2 %	(Committing Snapshot Effects: 0 Total)
0.9 ms 4.8 %	0.9 ms 4.8 %	(Committing Host Effects: 1 Total)
0.5 ms 2.8 %	2.9 ms 15.4 %	(Committing Changes)
0.4 ms 2.1 %	0.4 ms 2.1 %	DireBonjour [mount]
0.3 ms 1.6 %	0.3 ms 1.6 %	(Calling Lifecycle Methods: 0 Total)

Composant père

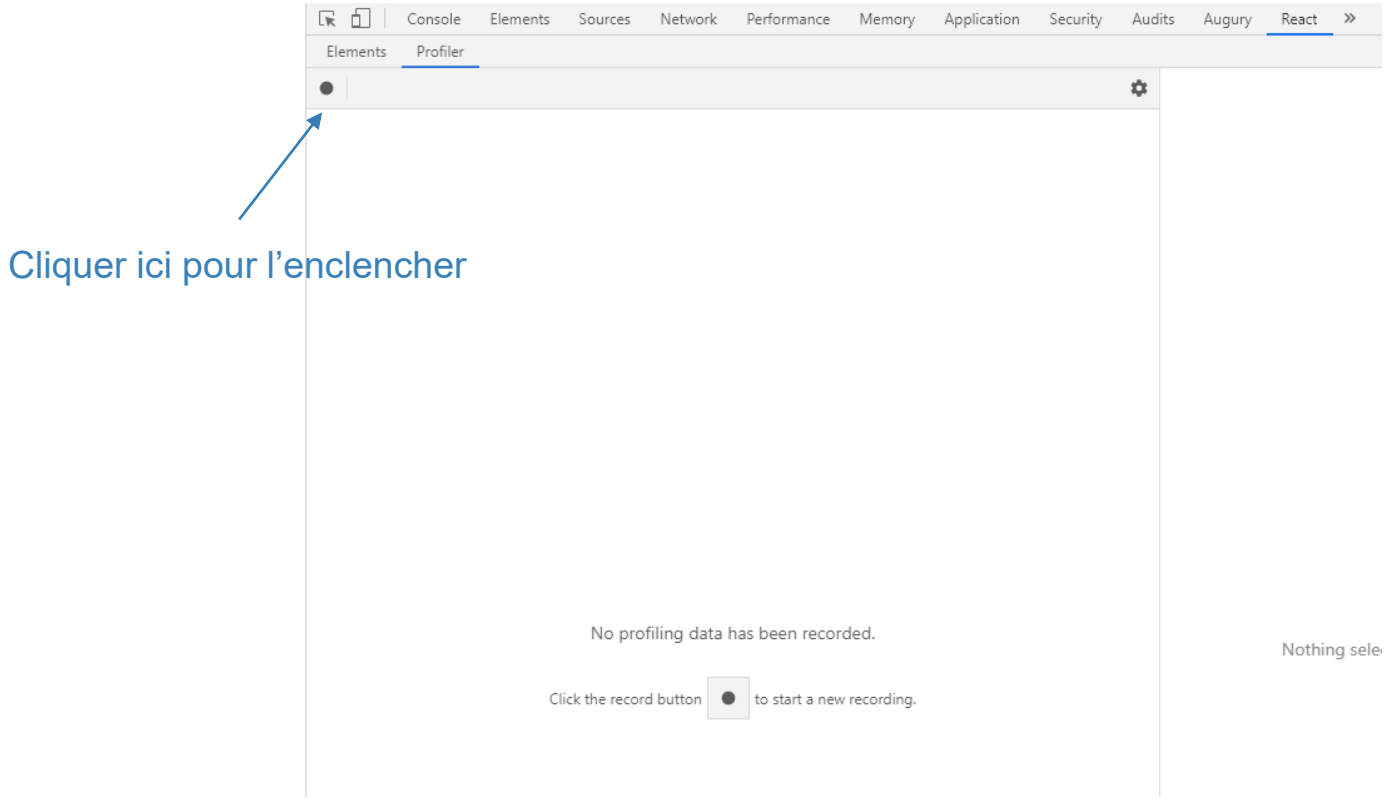
Mes 2 composants enfant

5.8 ms	30.9 %	Test [mount]
5.8 ms	30.9 %	App [mount]
5.8 ms	30.9 %	(React Tree Reconciliation: Completed Root)

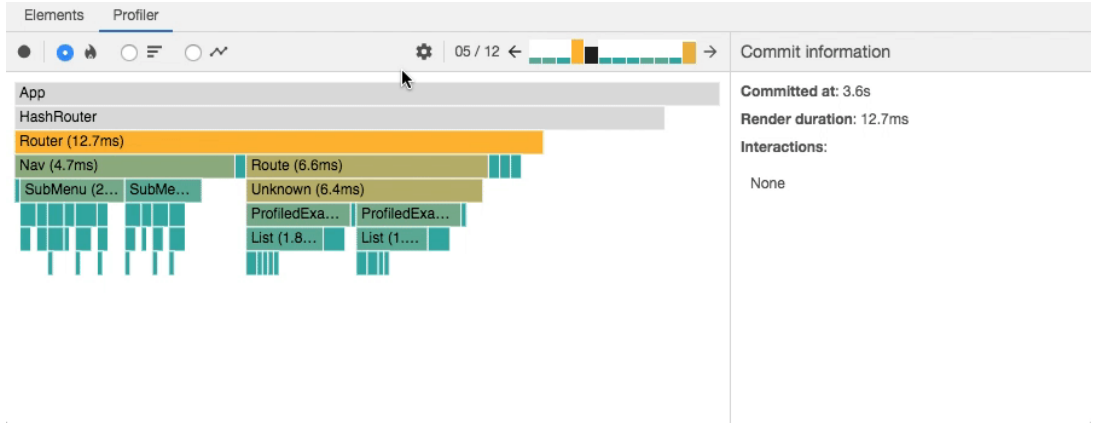
On voit ici que c'est principalement le ré-affichage du composant qui fut long

Mesurer les performances

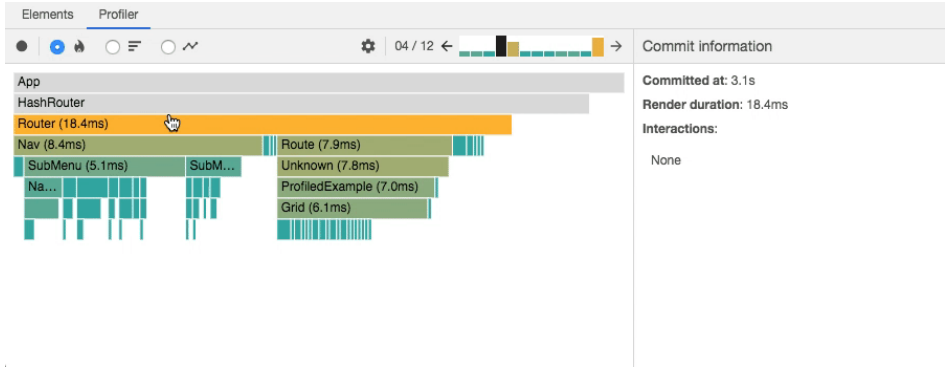
▷ Enfin, depuis peu, un profiler a été intégré directement dans le React Dev Tools :



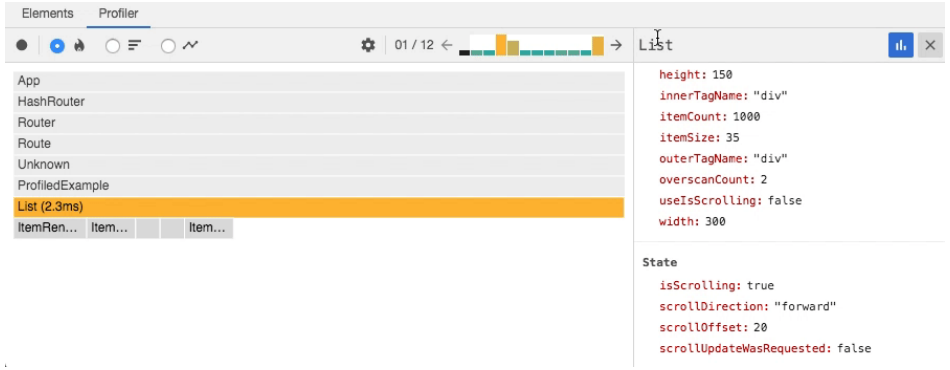
Mesurer les performances



Interface configurable

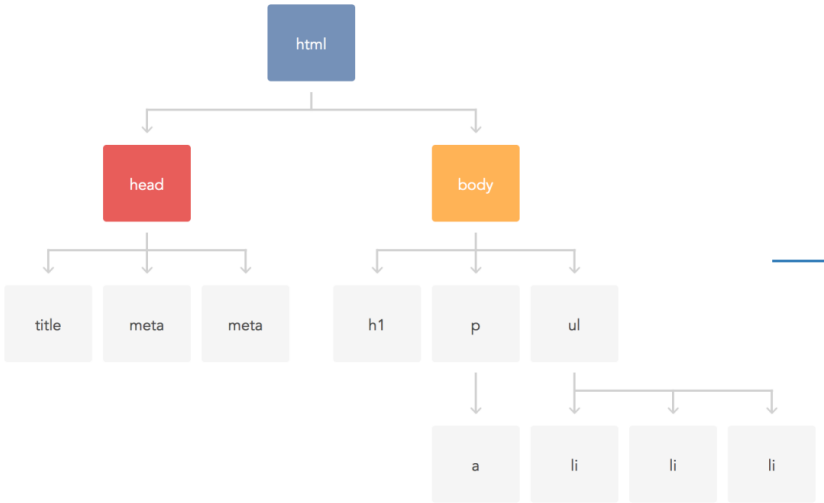


Consulter les valeurs des states



Naviguer entre les states successifs

Pour résumer

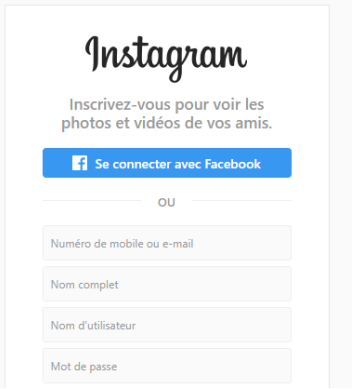


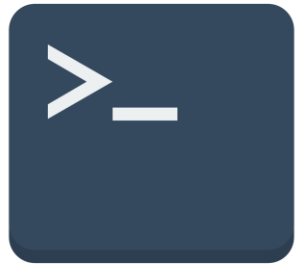
1. Votre code

```
React.createElement(
            "html",
            null,
            React.createElement("head", ...),
            React.createElement("body", ...)
);
```

2. Converti en VirtualDOM

3. Puis transformé en HTML, Swift ou même du bash





Questions ?