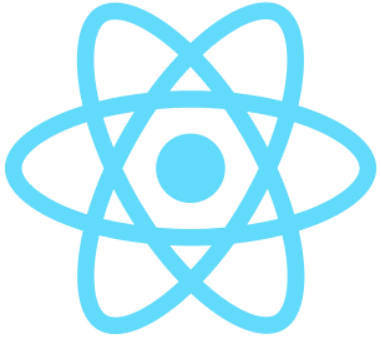
React



Première partie

Pré-requis

- Modèle client-serveur HTTP
- HTML
- DOM
- JavaScript

Quelques jalons dans l'histoire des technologies du Web

- **1993 HTML** : statique ou généré côté serveur
- **1995 JavaScript** : ajout d'interactions côté client
- **1996 CSS** : séparation structure / présentation
- **2004 AJAX** (requêtes HTTP depuis JS) ➟ modification des pages à partir de contenu serveur
- **2006 jQuery** : bibliothèque JS surcouche au DOM, manipulation facilitée des pages, plugins
- **2007 iPhone puis Android** : émergence d'un vrai web mobile
- **2010 Node.js + npm** : JS côté serveur, gestion de dépendances, croissance de l'écosystème
  ➟ JS comme environnement de développement mature
- **2014 HTML5** : version modernisée, nouveaux éléments sémantiques
- **2015 ECMAScript 6 (ES6)** : JS "moderne" (`let/const` , `modules` , classes, `async/await` , etc.)

Et tout du long, évolution des navigateurs et enrichissement des standards.

SPA : *Simple Page Application*

Par opposition aux architectures classiques où chaque page est générée en HTML côté serveur

- "Page" HTML squelette unique + code applicatif entier en JavaScript côté client
- Interrogation d'une API sur le serveur via des appels HTTP depuis le JS
- Modification dynamique du DOM
- Interception des navigations d'une url à l'autre

UX (*User eXperience*) "fluide", adapté aux applications web très "dynamiques" :

- modification continue des données (ex : réseaux sociaux, plateformes vidéo)
- manipulations complexes en temps réel côté client (ex : suite bureautique)
- notion de page peu pertinente (ex : applications cartographiques)

⚠️ Tout n'a pas à être une SPA (ex : Wikipedia, un journal en ligne, un dépôt de code Git, etc.)

Développement des SPA

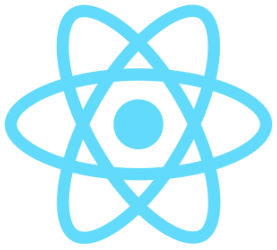
Modèle plébiscité mais pouvant se complefixier très vite :

- cohérence des données manipulées côté client (état)
- cohérence de l'interface par rapport à l'état
- gestion des urls et de la navigation interne (*routing client*)
- modularité
- performances
- etc.

Apparition rapide de nombreux frameworks JavaScript dédiés (Backbone.js, Ember, AngularJS, etc.)

- Architecture **M**odel-**V**iew-**C**ontroller
- Certains perçus comme trop dirigistes, peu performants, peu modulaires, etc.

React



Introduit par Facebook en 2014

Dédié au développement de SPA

Présenté à l'origine comme une **bibliothèque**, pas un framework

Se focalise sur la partie View de l'architecture MVC

Trés populaire, mais **pas un standard du web**

Principes :

- faciliter l'écriture de **composants** réutilisables
- garantir la **cohérence** de l'interface à chaque instant
- **modifier le DOM** de manière **performante et transparente**

Documentation officielle

Paradigmes de programmation

Programmation impérative

Description des étapes à suivre

Ex : "Quand le nom d'utilisateur change, le mettre à jour dans le header et dans la liste des messages"

V.S.

Programmation déclarative

Description du résultat voulu

Ex : "Dans le header et dans la liste des messages, afficher le nom de l'utilisateur courant"

en React, réalisé par ↓

Programmation réactive

Modification d'une source de données (état) ➠ propagation aux éléments qui en dépendent et mise-à-jour

Exemple élémentaire

Document HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <script src="example.js"></script>
  </head>
  <body>
    <h1>Exemple React</h1>
    <p>Éléments statiques</p>

    <!-- élément prévu pour React -->
    <div id="root"></div>
  </body>
</html>
```

Code Javascript

```javascript
import { createRoot } from 'react-dom/client';

// composant
function Hello(props) {
  return <h1>Hello {props.name} !</h1>
}

// "accrochage" d'un composant React
// au DOM de la page
const rootNode = document.getElementById('root');
const root = createRoot(rootNode);
root.render(<Hello name="Alice" />);
```

Une fois l'appel à `render()` effectué à la racine, tout ce qui est à l'intérieur va être géré par React.

Syntaxe JSX

Syntaxe proche du HTML, directement dans le code JavaScript, pour le *render*

- les composants sont assimilés à des éléments HTML
- les *props* sont assimilés à des attributs

Intérêts : rapidité d'écriture, encapsulation, rapprochement des concepts…

> ⚠️ Le JSX n'est pas du JavaScript valide ➟ repose sur une **transpilation**.
> En développement JS moderne, on travaille souvent dans un environnement transpilé.

Ce qu'on écrit

```
const message = 'Hello, world!';
const element = <h1 className="greeting">
  {message}
</h1>;
```

Transpilé en

```
const message = 'Hello, world!';
const element = React.createElement(
  "h1",
  { className: "greeting" },
  message
);
```

Syntaxe JSX

On mélange la notation balises/attributs avec des **expressions JavaScript entre accolades** `{ … }`

- Variables, valeurs calculées, appels de fonctions
- Structures de contrôles élémentaires (expressions booléennes et boucles `map`)

```
return (
  <div className={isOpened ? 'opened' : 'closed'}>
    <h3>Liste des utilisateurs</h3>
    <ul>
      {/* boucle */}
      {props.users.map(user =>
        <li key={user.id}>{formatUserName(user)}</li>
      )}
    </ul>
    {/* affichage conditionnel */}
    {errorMsg && <Error msg={errorMsg} />}
  </div>
)
```

Syntaxe JSX : quelques règles

- `false`, `null` et `undefined` ne sont pas rendus dans le DOM
- Toute balise doit être explicitement fermée (`<tag></tag>` ou `<tag />`)
- La casse de la première lettre des balises est importante :
  - **minuscule = balise HTML** (ex : `<h2>Introduction</h2>`)
  - **majuscule = composant React** (ex : `<UserName />`)
- Certains noms d'attributs HTML sont renommés :
  - `camelCase` : `onclick` → `onClick`, `tabindex` → `tabIndex`, etc.
  - attributs en conflit avec des mots-clés JS : `class` → `className`, `for` → `htmlFor`, etc.
- Les commentaires s'écrivent par blocs, aussi entre accolades `{/* … */}`

https://react.dev/learn/writing-markup-with-jsx

Déclaration de composant

En React, un composant est une **fonction** qui accepte un objet ***props*** (par convention, pour *"properties"*) et **retourne un fragment de HTML** exprimé en JSX.

```
function MyComponent(props) {
  return <h1>Hello {props.name} !</h1>;
}
```

Le même composant, écrit sous forme d'*arrow function* et en destructurant les *props*.

```
const MyComponent = ({ name }) => <h1>Hello {name} !</h1>;
```

⚠️ Vous pourrez rencontrer une façon plus ancienne de déclarer les composants en tant que classes. Cette approche est maintenant obsolète, **on verra ici uniquement les composants fonctionnels**.

Composants et *props*

Une **application React** est constituée d'un **arbre de composants** :

- Chaque composant peut **rendre** du HTML et/ou appeler d'autres composants
- Les données se propagent dans l'arbre via les ***props***
- Les données se propagent donc uniquement **de haut en bas**
- Un composant est **re-rendu** automatiquement quand il reçoit de **nouvelles valeurs de *props***
- Les composants dont les *props* n'ont pas changé ne sont pas re-rendus

https://react.dev/learn/passing-props-to-a-component

Virtual DOM

Modifier le contenu d'une page dynamiquement en JS nécessite de passer par l'API du **DOM**.

Par exemple :

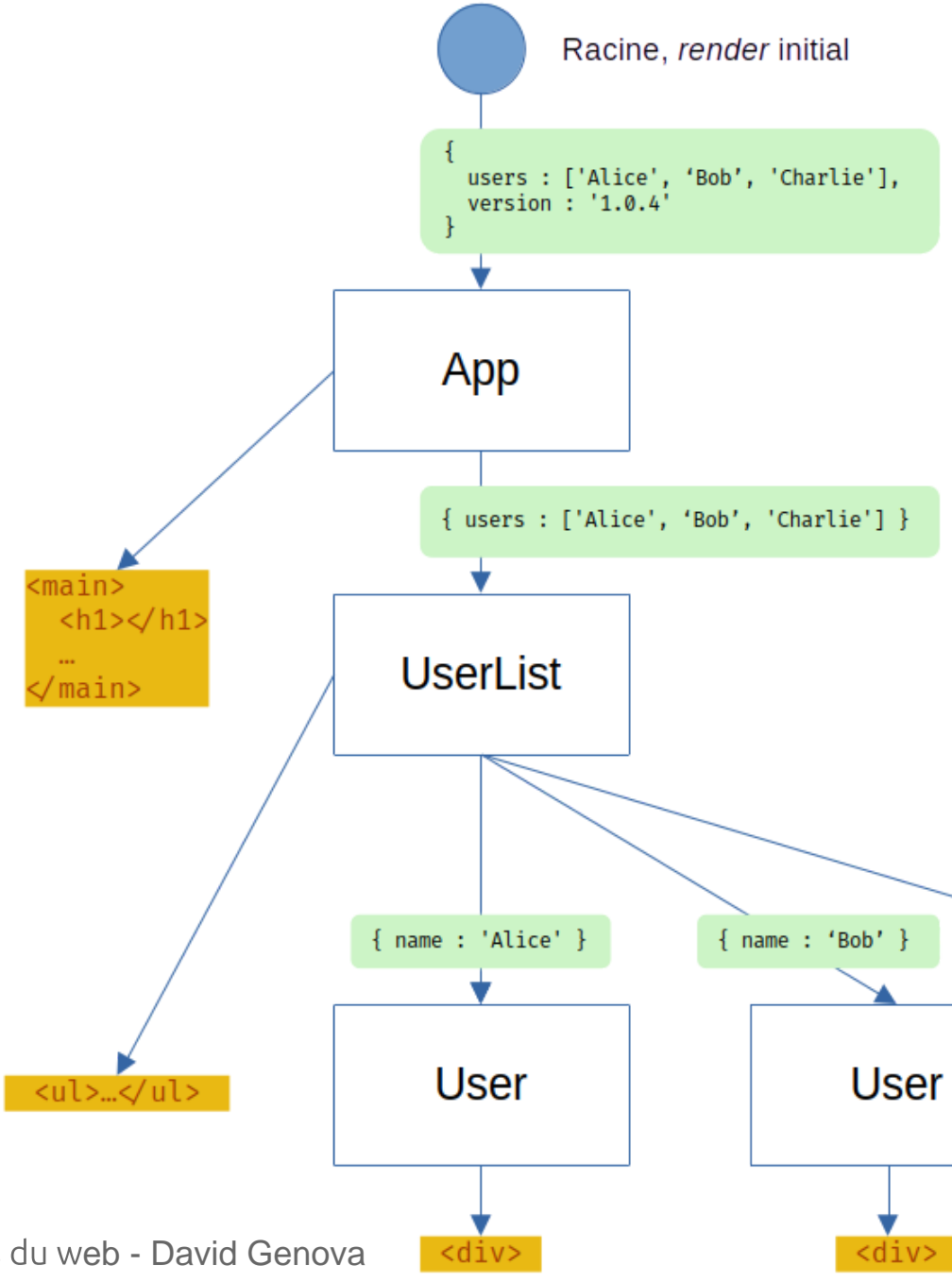
```
const myMessage = document.createElement('div');
myMessage.className = 'message';
myMessage.innerText = 'Lorem Ipsum';

const messageList = document.getElementById('messageList');
messageList.appendChild(myMessage);
```

React introduit une notion de **"DOM virtuel"** pour :

- **abstraire ces appels** à travers la syntaxe JSX
- **optimiser les accès au DOM réel** en appliquant des mises à jour minimales

```jsx
// Un composant prend en argument un unique objet "props"
function User(props) {
  // ce composant reçoit la prop 'name'
  return <li>Hello, {props.name}</li>;
}

// La syntaxe de destructuration d'objet fait mieux apparaître les props en entrée
function UserList({ users }) {
  return <ul>
    {users.map(user => <User name={user.name} key={user.id} />)}
  </ul>;
}

function App({ users, version }) {
  return <main>
    <h1>Welcome to version {version} of the app!</h1>
    <UserList users={users} />
  </main>;
}

const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];

createRoot(document.getElementById('root')).render(
  <App users={users} version="1.0.4" />
);
```

ℹ️ Dans une vraie application, on préférera créer un fichier par composant.



- Chaque composant rend du HTML et/ou d'autres composants
- Les *props* se propagent de haut en bas dans l'arbre

Props

On peut passer ce qu'on veut en *props* :

- nombres, chaînes de caractères, booléens, objets, tableaux, etc.
- fonctions *callbacks* (gestionnaires d'événements notamment, abordés plus loin)
- du JSX, une instance de composant, un autre composant React

Une *prop* spéciale : `children` ➟ permet de passer sa valeur en contenu de la balise JSX.

```
const Details = ({ title, children }) => {
  return <details>
    <summary>{title}</summary>
    {children}
  </details>;
}

const App = () => {
  return <Details title="Conditions d'utilisation">
    <p>Article 1. : blabla</p>
    <p>Article 2. : …</p>
  </Details>
}
```

State

Matérialise l'**état interne** d'un composant, sa "**mémoire**"

- Privé et encapsulé ➟ on doit passer par les *props* pour propager ce *state* aux enfants
- Persistant entre deux rendus
- Quand le *state* d'un composant change, le composant est automatiquement re-rendu

⚠️ Une des grandes questions d'une appli React est de savoir quel composant doit gérer quel *state*.

- Un composant sans état (*stateless*) sera plus réutilisable (pas de comportement embarqué, le rendu dépendra uniquement des *props*)
- Partager un état entre deux composants = en faire les descendants d'un même composant *stateful*

https://react.dev/learn/state-a-components-memory

Illustration du *state* : un composant d'horloge

```
import { useState, useEffect } from 'react';

const Clock = () => {
  // useState() prend la valeur initiale et retourne
  // [valeurCourante, fonctionDeModification]
  const [date, setDate] = useState(new Date());

  // mise à jour périodique
  useEffect(() => {
    const timerID = setInterval(      // fonction JavaScript native
      () => { setDate(new Date()); }, // callback
      1000                            // délai de 1 seconde
    );

    return () => { clearInterval(timerID); }
  }, []);

  return <div>Il est {date.toLocaleTimeString()}.</div>
}
```

ℹ️ `useState` et `useEffect` font parties des **hooks**, qu'on détaillera dans la seconde partie.

Interactions utilisateurs

On attache des *callbacks* aux différents types d'événements via les attributs JSX dédiés : `onClick`, `onKeyPress`, `onFocus`, `onSubmit`, … (version `camelCase` de ceux du HTML)

```
function EventExample() {
  function handleClickButton() {
    console.log('Bouton cliqué !')
  }

  function handleInputChange(event) {
    console.log("Valeur de l'input : ", event.target.value);
  }

  return <>
    <button type="button" onClick={handleClickButton}>Click!</button>
    <input type="text" onChange={handleInputChange} />
  </>;
}
```

Interactivité = *state* + *events*

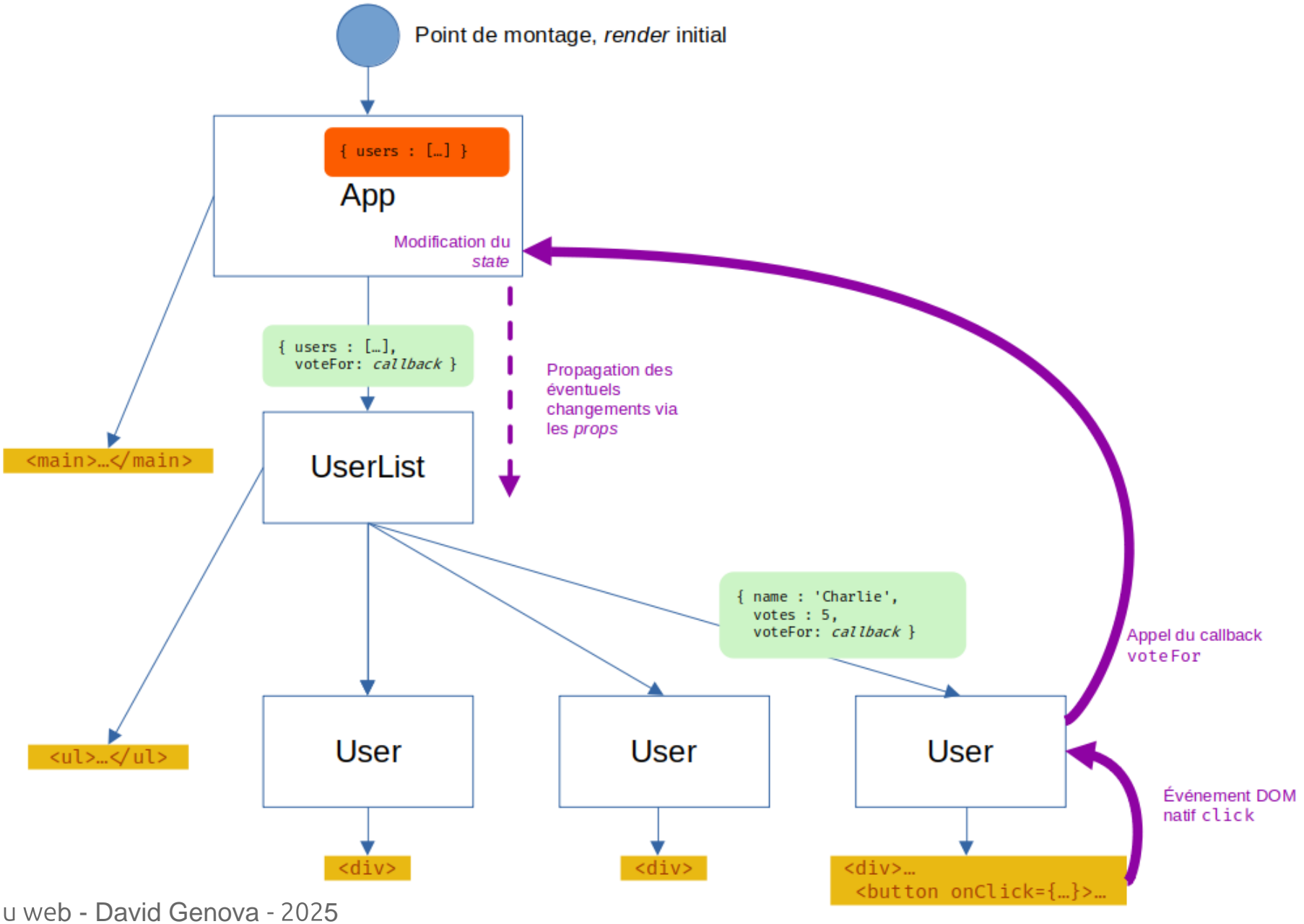
On crée des composants interactifs en modifiant leur *state* via des événements utilisateurs.

```
import { useState } from 'react';

function Clicker() {
  const [nbClicks, setNbClicks] = useState(0);

  const incrementNbClicks = () => {
    setNbClicks(nbClicks + 1);
  }

  return <div>
    <p>Nombre de clics : {nbClicks}</p>
    <button type="button" onClick={incrementNbClicks}>Click!</button>
  </div>;
}
```

Passage de callbacks en *props*

→ Permet la remontée d'informations aux parents en respectant le flux de propagation descendant.

```
// Composant stateless, prend le callback onChangeTheme en prop
function ThemeSwitch({ currentTheme, onChangeTheme }) {
  return <div>
    <button
      disabled={currentTheme === 'dark'}
      onClick={() => { onChangeTheme('dark'); }}>Dark</button>
    <button
      disabled={currentTheme === 'light'}
      onClick={() => { onChangeTheme('light'); }}>Light</button>
  </div>;
}

function App() {
  const [theme, setTheme] = useState('light'); // L'état est maintenu ici

  return <div>
    <ThemeSwitch currentTheme={theme} onChangeTheme={setTheme} />
    <Content theme={theme} />
  </div>;
}
```


Point de montage, *render* initial
{ users : […] }
App
Modification du *state*
{ users : […],
voteFor: *callback* }
Propagation des éventuels changements via les *props*
<main>…</main>
UserList
{ name : 'Charlie',
votes : 5,
voteFor: *callback* }
Appel du callback voteFor
…
User
User
User
Événement DOM natif click

…
<button onClick={…}>…

Résumé

- Une application web React est constituée d'un **arbre de composants**
- Les composants **rendent des fragments de HTML**, gérés dans le DOM par React
- L'**état de l'application est maintenu dans le *state*** de certains composants
- Les **interactions utilisateur** modifient ces *states*
- Les changements de *state* sont **propagés via les *props* de haut en bas** dans l'arbre
- Chaque **changement de *state/props* entraîne le re-rendu** des composants qui en dépendent

Liens utiles

- Site officiel React
- React Developer Tools
  - Firefox
  - Chrome, Chromium, Edge
- Create React App
  Pour monter un environnement de développement avec un outillage pré-installé (utilisé en TME)

Suite dans la deuxième partie :

Hooks et cycle de vie, connexion avec API, formulaires.