formation.tech

React

formation.tech

Introduction

Introduction - Historique

- React est une bibliothèque permettant de simplifier la création d'interface utilisateur
- Imaginée par Jordan Walke chez Facebook en 2011 rejoint rapidement par d'autres ingénieurs Facebook
- D'abord conçue pour le navigateur, il est aujourd'hui possible de l'utiliser côté serveur, pour créer des interfaces graphiques dans des applications natives (iOS, Android, Windows...) et même plus (programmes en ligne de commandes...)
- Rendue Open-Source en 2013 (d'abord sous Licence BSD avec Clause puis sous Licence MIT en novembre 2017)
- Lors de sa présentation à la JSConf en 2013 React a fait un flop car ses concepts novateurs était incompris
- Voir la vidéo React.js: The Documentary https://www.youtube.com/watch?v=8pDqJVdNa44&t=3433s

Introduction - Qui utilise React ?

- Une des premières utilisation de React chez Facebook en production : la barre de likes et commentaires sous les posts
- Puis Facebook Chat (devenu Messenger) et Instagram Web (suite au rachat) sont devenues les premières applications entièrement écrites avec React
- Aujourd'hui

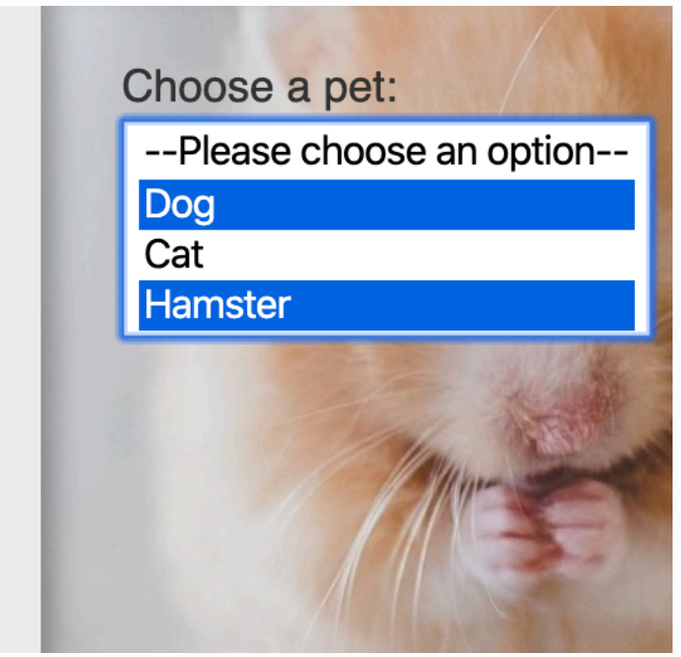
- Yahoo! Mail
- AirBnb
- Netflix
- Dropbox
- SNCF Connect
- formation.tech
- et bien d'autres... (React est la bibliothèque UI la plus téléchargée actuellement)

Introduction - Qu'est-ce qu'un composant ?

- Un composant est un moyen simple et isolé de regrouper :
  - Du HTML pour la structure de données
  - Du CSS pour la mise en forme
  - Du JavaScript pour le comportement
- Exemple : la balise select

https://developer.mozilla.org/fr/docs/Web/HTML/Element/select

```
<label for="pet-select">Choose a pet:</label>

<select name="pets" id="pet-select" multiple>
    <option value="">--Please choose an option--</option>
    <option value="dog">Dog</option>
    <option value="cat">Cat</option>
    <option value="hamster">Hamster</option>
    <option value="parrot">Parrot</option>
    <option value="spider">Spider</option>
    <option value="goldfish">Goldfish</option>
</select>
```



Introduction - Qu'est-ce qu'un composant ?

- Le composant Select de react-select
  https://react-select.com/

```jsx
import React from 'react';

import Select from 'react-select';
import { colourOptions } from '../data';

export default () => (
  <Select
    defaultValue={[colourOptions[2], colourOptions[3]]}
    isMulti
    name="colors"
    options={colourOptions}
    classNamePrefix="select"
  />
);
```

Purple × Red ×

Introduction - Ecosystème

- A la différence de Google qui maintient tout un écosystème de bibliothèques dans Angular (pour gérer les requêtes HTTP, les formulaires, les pages ou les animations), React ne propose qu'un nombre limité de bibliothèques :
  - react
    Bibliothèque permettant la création de composants et la communication
  - react-dom/client
    Permet de faire le rendu d'un composant dans le navigateur via l'API DOM
  - react-dom/server
    Permet de faire le rendu d'un composant dans Node.js
  - react-native
    Permet de faire le rendu d'un composant dans une application native iOS ou Android

Introduction - Documentation

- Doc officielle
  https://react.dev/
- Doc Legacy
  https://legacy.reactjs.org/
- Tutoriel Officiel
  https://react.dev/learn/tutorial-tic-tac-toe
- Blog
  https://react.dev/blog
- TypeScript
  https://react.dev/learn/typescript
  https://react-typescript-cheatsheet.netlify.app/
  https://github.com/piotrwitek/react-redux-typescript-guide#readme

Introduction - Comment utiliser React ?

- On peut techniquement importer React via une balise script

```
<body>
  <div id="root"></div>
  <script src="../node_modules/react/umd/react.development.js"></script>
  <script src="../node_modules/react-dom/umd/react-dom.development.js"></script>
  <script src="./src/main.js" type="module"></script>
</body>
```

- Mais cela nous empêcherait d'utiliser la syntaxe JSX ou obligerait le navigateur à transformer lui même le JSX (peu performant)

Introduction - Comment utiliser React ?

- Pour utiliser React on peut soit :
  - utiliser une toolchain (une suite d'outils)
  - utiliser un framework
- Les toolchains principales :
  - create-react-app (officielle mais plus maintenue)
  - Vite : https://vitejs.dev/
  - Parcel : https://parceljs.org/
  - Nx : https://nx.dev/recipes/react
- Les frameworks principaux :
  - Next.js : https://nextjs.org/
  - Remix : https://remix.run/
  - Gatsby : https://www.gatsbyjs.com/

Introduction - Virtual DOM, diffing et réconciliation

- Lorsque React devient Open Source en 2013 la plupart des bibliothèques UI populaires (Backbone.js, AngularJS, Ember.js, Knockout.js...) sont basées sur le concept de data binding
- Principe du data binding : stocker les données qui doit s'afficher ou décrivent l'interface graphique dans un objet (Model) et écouter les changements de cette objet pour patcher la vue (View) et inversement
- Avec React l'approche est plutôt de décrire entièrement l'UI en fonction du modèle (State) et de re-rendre (rerender) l'intégralité de l'UI lorsqu'un changement intervient au niveau du modèle
- L'approche peut sembler contre-performante en y pensant mais React arrive à un très bon niveau de performance grâce à un mécanisme appelé le Virtual DOM

Introduction - DOM

- Voici un composant écrit avec l'API DOM qu'on va afficher chaque seconde sur une page web
- Alors que seule la date change, l'ensemble des éléments du DOM seront recréés et redessinés par le navigateur

```
import { createRoot } from './similar-to-react.js';

function App() {
  const name = 'Romain';
  const date = new Date();

  const pEl = document.createElement('p');
  pEl.append('Hello ', name);

  const divEl = document.createElement('div');
  divEl.append(date.toLocaleTimeString());

  return [pEl, divEl];
}

const root = createRoot(document.getElementById('root'));
root.render(App());

setInterval(() => {
  root.render(App());
}, 1000);
```

```
<body>
▼<div id="root"> == $0
  ▶<p> ••• </p>
  <div>10:24:50</div>
```

Hello Romain

10:27:04

Console What's New Rendering × Sens

Paint flashing
Highlights areas of the page (green) that need to be
to photosensitive epilepsy.

Introduction - Virtual DOM

- Avec React, on construit un arbre d'éléments React appelé Virtual DOM

```
function App() {
  const name = 'Romain';
  const date = new Date();

  return React.createElement(
    React.Fragment,
    null,
    React.createElement('p', null, 'Hello ', name),
    React.createElement('p', null, date.toLocaleTimeString()),
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(App());

setInterval(() => {
  root.render(App());
}, 1000);
```

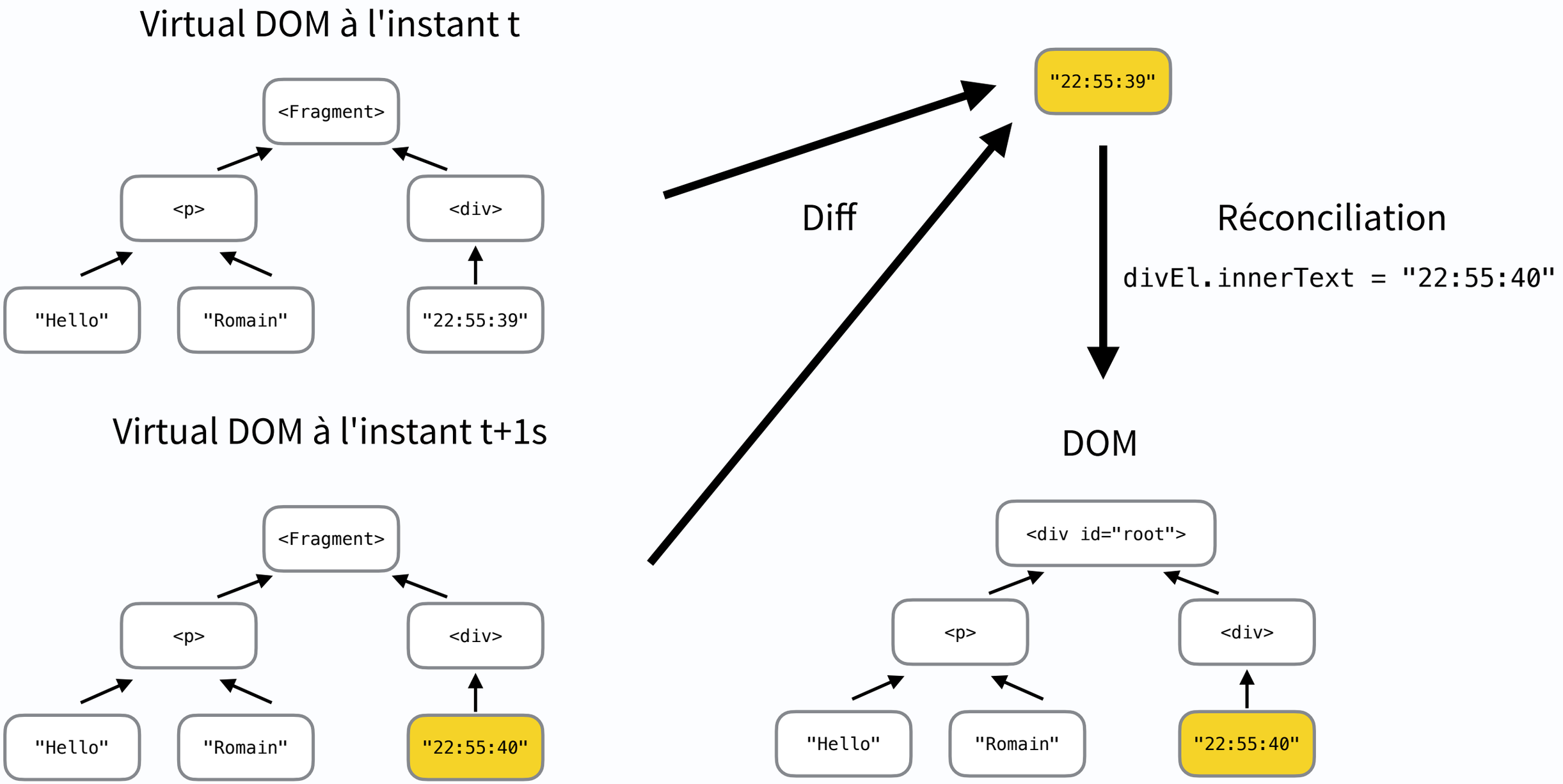
```
▼ <body>
  ▼ <div id="root">
    ▼ <p> == $0
        "Hello "
        "Romain"
      </p>
      <p>10:30:41</p>
```

Hello Romain

10:30:45

- Au moment de faire le rendu chaque seconde, seule le noeud de texte contenant la date sera redessiné :

Introduction - Diff et réconciliation

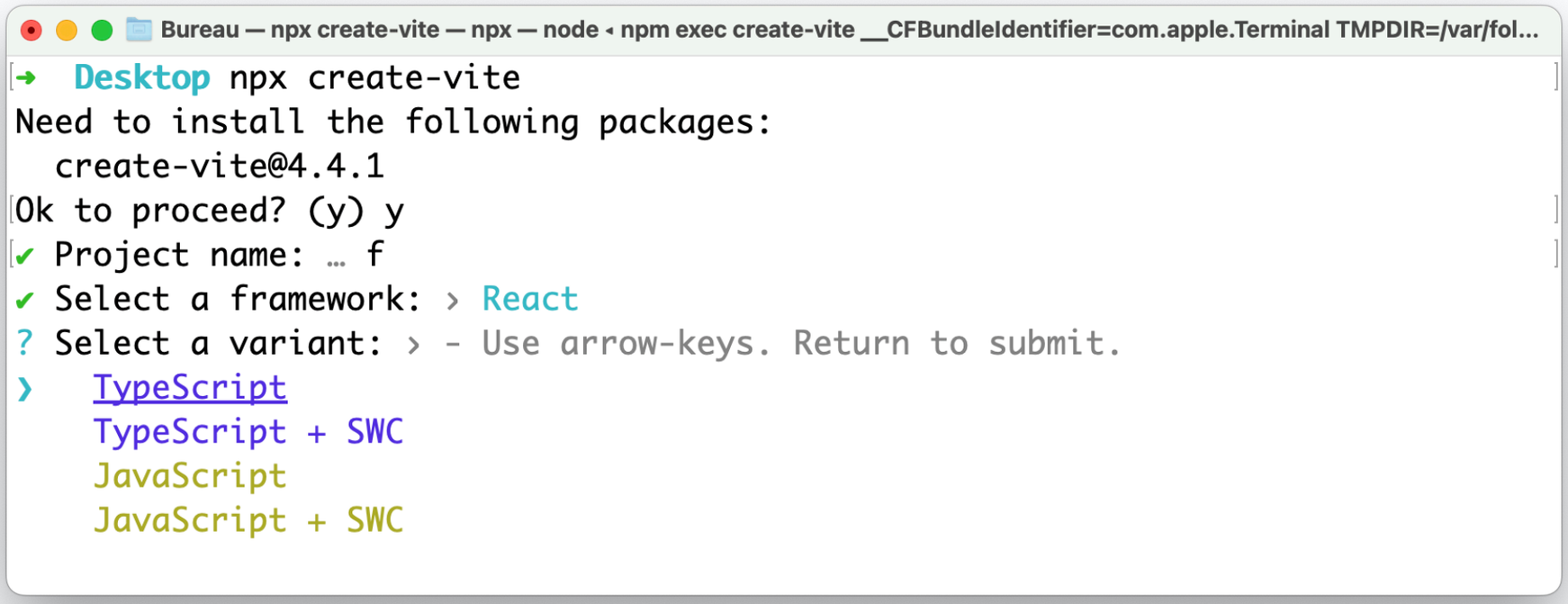


Introduction - Unopinionated

- React se dit "unopinionated" (non dogmatique) c'est à dire qu'il ne cherche pas à imposer ses choix à ses utilisateurs (gestion des formulaires, des routes, architecture de fichiers)
- Néanmoins il est important de faire ses propres choix et que ces choix soit communiqués avec les autres développeurs du projets (i.e. écrire des conventions)

Introduction - Installation

- Pour démarrer rapidement avec React il nous faut :
  - une version récente de Node.js
  - une IDE qui supporte le JSX (Visual Studio Code, WebStorm...)
- Pour démarrer rapidement je conseille d'utiliser la toolchain Vite :
  `npx create-vite`



Introduction - Installation

- Sans create-vite, il faut à minima installer React et ReactDOM :
  ```
  npm install react react-dom
  ```
- Vite
  ```
  npm install vite --save-dev
  ```

Introduction - Installation TypeScript

- Si on souhaite utiliser TypeScript il faut installer également installer :
  - le paquet typescript (pas obligatoire avec Vite mais préférable)
  - les déclarations TypeScript de React et ReactDOM (qui sont 2 projets JavaScript)
- `npm i typescript @types/react @types/react-dom -D`
- En interne Vite utilise esbuild qui ne va pas vérifier les types, installé séparément le paquet TypeScript sera utilisé par :
  - l'IDE (via tsserver)
  - Au moment de build de prod (via tsc qui sera lancé en plus de vite build)
- En TypeScript si on écrit du JSX, l'extension des fichiers doit obligatoirement être *.tsx*

Introduction - Point d'entrée HTML

- Avec Vite, le point d'entrée de l'application est index.html, il doit être à la racine du projet
- Vite analyse les balises script et link pour déterminer les fichiers à builder

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="src/main.js" type="module"></script>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

Introduction - Point d'entrée JS

- Le fichier principal React contiendra à minima :

```jsx
import App from './App';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(<App />);
```

- En TypeScript

```tsx
import App from './App';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root') as HTMLElement);
root.render(<App />);
```

Introduction - Mode Strict

- Les templates React utilisent en général le mode strict de React avec le composant StrictMode :

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import App from './App';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>,
);
```

Introduction - Mode Strict

- En mode Strict :
  - Les composants seront rendus 2 fois pour détecter des problèmes de composant impurs
  - Les effets seront exécutés 2 fois pour détecter des problèmes de nettoyage (clearInterval, removeEventListener...)
  - React vérifiera que vous n'utilisez pas d'API legacy (peu probable si vous démarrez)
- Attention cela peut vous perturber si vous utiliser les DevTools du navigateur :
  - les requêtes HTTP exécutez au chargement du composant s'exécuteront 2 fois
  - les logs dans les composants seront doublés
- Documentation https://react.dev/reference/react/StrictMode

Introduction - Vite

- Avec vite on utilisera les commande suivantes

```
"scripts": {
  "dev": "vite",
  "build": "tsc --noEmit && vite build",
  "prebundle": "vite optimize",
  "preview": "vite preview"
}
```

- vite
  Pour lancer le dev server

- vite optimize
  Pour accélérer le dev server en transformant les modules commonjs ou UMD en ESM du répertoire node_modules

- vite build
  Pour créer le build de prod (lancer également tsc pour vérifier les types)

- vite preview
  Pour servir les fichiers buildés par vite build

formation.tech

JSX

JSX - Introduction

- JSX est une syntaxe inventée par Facebook pour React qui étendre la syntaxe JavaScript
- Dans React le JSX permet de créer un arbre d'éléments React en mémoire appelé Virtual DOM
- Il permet pour des applications JavaScript de programmer l'interface de façon déclarative comme en HTML
- La syntaxe utilisée est XML (attention aux balises vide comme `<img />`)
- Le JSX n'a pas pour vocation a être implémenté par les navigateurs mais d'être transformé en JavaScript par un transpileur (Babel, esbuild...)
- D'autres projets utilisent également JSX aujourd'hui : Vue.js, preact, Solid, Qwik...

JSX - Transpiler

- Les transpilateurs Babel et esbuild supportent la syntaxe JSX
- Avec Babel il faut utiliser le plugin @babel/plugin-transform-react-jsx
- Les transpilateurs ont 2 modes de fonctionnement :
  - classic, qui transforme les balises JSX en *React.createElement* (React devra être importé)
  - automatic (recommandé), qui transforme les balises JSX en *jsxs* (jsxs est importé automatiquement)

JSX - Runtime classic

- Avec le runtime classic ce code

```
function App() {
  const now = new Date();
  const prenom = 'Romain';

  return (
    <div className="App" id="App">
      <p>Heure : {now.toLocaleTimeString()}</p>
      <p>Prénom : {prenom}</p>
    </div>
  );
}

export default App;
```

- Devient

```
function App() {
  const now = new Date();
  const prenom = 'Romain';

  return (
    React.createElement('div', { className: 'App', id: 'App' },
      React.createElement('p', null, 'Heure : ', now.toLocaleTimeString()),
      React.createElement('p', null, 'Prénom : ', prenom)
    )
  );
}

export default App;
```

JSX - Runtime classic

- Avec le runtime classic, React n'est pas importé automatiquement, il doit être importé à nous de l'inclure
- S'il n'est pas inclut l'erreur suivante se produit :
  React Must Be in Scope When Using JSX
- Il faut donc un import de React même s'il n'était pas explicitement utilisé :

```jsx
import React from 'react'; // obligatoire en runtime classic (par défaut avant React 17)

function App() {
  const now = new Date();
  const prenom = 'Romain';

  return (
    <div className="App" id="App">
      <p>Heure : {now.toLocaleTimeString()}</p>
      <p>Prénom : {prenom}</p>
    </div>
  );
}

export default App;
```

JSX - Runtime automatic

- Avec le runtime automatic :

```
import { jsxs as _jsxs } from 'react/jsx-runtime';

function App() {
  const now = new Date();
  const prenom = 'Romain';

  return _jsxs('div', {
    className: 'App',
    id: 'App',
    children: [
      _jsxs('p', { children: ['Heure : ', now.toLocaleTimeString()] }),
      _jsxs('p', { children: ['Prénom : ', prenom] }),
    ],
  });
}

export default App;
```

- Plus besoin d'inclure React car la fonction jsxs est incluse automatiquement
- Le build est plus petit car on importe qu'une partie de React (react/jsx-runtime)

JSX - Accolades

- En JSX, on utilise les accolades pour utiliser des expressions JavaScript
- Les accolades créées des noeuds de texte si on utilise les types string, number ou React.Element :

```
function WillRender() {
  const name = 'Romain';
  const age = 38;
  const element = <div>React Element</div>;

  return (
    <div className="App">
      <p>Hello {name}, I'm {age}</p>
      {element}
    </div>
  );
}
```

- Les types boolean, null ou undefined peuvent être utilisés mais il ne créent pas de noeud

```
function WillDoNothing() {
  return (
    <div className="App">
      {undefined} {null} {false} {true}
    </div>
  );
}
```

JSX - Accolades

- On peut également utiliser des types itérables qui vont créer plusieurs noeuds

```
function WillLoop() {
  return (
    <div className="App">
      {['ABC', 123, <div>Element</div>]}
    </div>
  );
}
```

- Le mêmes comportement s'applique :
  - les types string, number et React.element s'affichent
  - les types boolean, null et undefined n'affichent rien

JSX - Accolades

- Attention les objets autres que string, React.element et Iterable provoquent une erreur :

```
function WillThrow() {
  const coords = { x: 1, y: 2};
  const now = new Date();
  return (
    <div className="App">
      {coords} {now}
    </div>
  );
}
```

← Erreur : *Objects Are Not Valid as a React Child*

JSX - Accolades

- Les commentaires s'écrivent de la manière suivante (valeur === undefined) :
  `{/* Mon commentaire */}`

```
function Comment() {
  return (
    <div className="App">
      {/* Comment */}
    </div>
  );
}
```

JSX - Accolades

- En TypeScript on peut utiliser le type ReactNode pour typer les expressions utilisable avec les accolades JSX :

```tsx
function App() {
  const name: ReactNode = 'Romain';
  const age: ReactNode = 38;
  const element: ReactNode = <div>React Element</div>;

  return (
    <div className="App">
      <p>Hello {name}, I'm {age}</p>
      {element}
    </div>
  );
}
```

- Le type ReactNode dans @types/react :

```tsx
type ReactNode =
  | ReactElement
  | string
  | number
  | Iterable<ReactNode>
  | ReactPortal
  | boolean
  | null
  | undefined
;
```

JSX - Attributs JSX

- Les attributs JSX sont passés en 2ème paramètre de React.createElement

```jsx
// En JSX :
<input type="text" maxLength={10} required />

// Une fois buildé :
React.createElement('input', {
  type: 'text',
  maxLength: 10,
  required: true,
}),
```

```jsx
// En JSX :
React.createElement(Hello, {
  name: 'Romain',
  age: 38,
  isActive: true,
})

// Une fois buildé :
<Hello name="Romain" age={38} isActive />
```

- Lorsqu'on passe une constante de type string (ici "text" ou "Romain") les accolades sont optionnelles
- Lorsqu'on ne passe pas de valeur (ici required ou isActive) cela revient à passer true

JSX - Attributs JSX

- On peut également passer directement un object contenant plusieurs attributs JSX avec la syntaxe :
  `{...obj}`

```
function Attributes() {
  const inputProps = {
    type: 'text',
    maxLength: 10,
  };
  return (
    <div className="App">
      <input {...inputProps} required />
    </div>
  );
}
```

JSX - Attributs JSX

- En TypeScript le type de l'objet dépend de l'élément :
  - Pour une div (et les éléments neutres comme <header>, <footer> `HTMLAttributes<HTMLDivElement>`
  - Pour un élément plus spécifique XXXHTMLAttributes<HTMLXXXElement> : `InputHTMLAttributes<HTMLInputElement>` `FormHTMLAttributes<HTMLFormElement>` `VideoHTMLAttributes<HTMLVideoElement>`

```tsx
import { InputHTMLAttributes } from 'react';

function Attributes() {
  const inputProps: InputHTMLAttributes<HTMLInputElement> = {
    type: 'text',
    maxLength: 10,
  };

  return (
    <div className="App">
      <input {...inputProps} required />
    </div>
  );
}

export default Attributes;
```

JSX - Rendu conditionnel

- Pour rendre des éléments de manière conditionnelle on réutilise ce qu'on a appris à propos des accolades JSX

```jsx
function App() {
  let element;

  if (condition) {
    element = <div>If true</div>
  }

  return (
    <div className="App">
      {element}
    </div>
  );
}
```

```jsx
function App() {
  let element;

  if (condition) {
    element = <div>If true</div>
  } else {
    element = <div>If not</div>
  }

  return (
    <div className="App">
      {element}
    </div>
  );
}
```

- Un élément React sera affiché, undefined sera ignoré

JSX - Rendu conditionnel

- En TypeScript

```tsx
function App() {
  let element: ReactNode;

  if (condition) {
    element = <div>If true</div>
  }

  return (
    <div className="App">
      {element}
    </div>
  );
}
```

```tsx
function App() {
  let element: ReactNode;

  if (condition) {
    element = <div>If true</div>
  } else {
    element = <div>If not</div>
  }

  return (
    <div className="App">
      {element}
    </div>
  );
}
```

JSX - Rendu conditionnel

- On peut également utiliser des expressions conditionnelles

```
function App() {
  const element = condition && <div>If true</div>;

  return (
    <div className="App">
      {element}
    </div>
  );
}
```

```
function App() {
  return (
    <div className="App">
      {condition && <div>If true</div>}
    </div>
  );
}
```

JSX - Rendu conditionnel

- On peut également utiliser des expressions conditionnelles

```
function App() {
  const element = condition ? <div>If true</div> : <div>If not</div>;

  return (
    <div className="App">
      {element}
    </div>
  );
}
```

```
function App() {
  return (
    <div className="App">
      {condition ? <div>If true</div> : <div>If not</div>}
    </div>
  );
}
```

JSX - Listes

- Pour afficher plusieurs éléments il suffit d'utiliser un tableau

```
function App() {
  const names = ['Romain', 'Edouard'];
  const elements = [];

  for (const name of names) {
    elements.push(<div key="name">{name}</div>)
  }

  return (
    <div className="App">
      {elements}
    </div>
  );
}
```

- L'attribut key est optionnel, en dev un warning s'affichera dans la console s'il n'est pas présent
- En prod le warning disparait

JSX - Listes

- L'attribut key sert à React à améliorer le diffing et mieux déterminer les changements apportés à des listes
- La valeur passée doit être unique
- L'attribut key s'utilise sur les éléments à la racine du tableau uniquement
- La valeur doit être la plus stable possible dans le temps :
  - l'élément du tableau d'origine si les valeurs ne sont pas éditables et uniques
  - l'indice du tableau d'origine si on ne peut pas le trier ou supprimer des éléments
  - l'id est idéal si tableau d'enregistrement provenant de la base de données
  - si besoin de générer une valeur (uniq(), uuid(), Math.random()) ne pas le faire à l'intérieur du composant (au prochain rendu, la valeur changera)

JSX - Listes

- Comme avec les listes on peut créer le tableau en une seule expression en utilisant .map

```
function App() {
  const names = ['Romain', 'Edouard'];
  const elements = names.map((name) => <div key="name">{name}</div>);

  return (
    <div className="App">
      {elements}
    </div>
  );
}
```

- Avec .map chaque élément du tableau est placé dans un nouveau tableau en étant transformé par la fonction de transformation :

```
[                                                                          [
  'Romain',  ⟶  (name) => <div key="name">{name}</div>  ⟶                    <div key="name">Romain</div>,
  'Edouard',                                                               <div key="name">Edouard</div>,
]                                                                          ]
```

JSX - Listes

- Utilisation de .map directement en JSX

```
function App() {
  const names = ['Romain', 'Edouard'];

  return (
    <div className="App">
      {names.map((name) => <div key="name">{name}</div>)}
    </div>
  );
}
```

JSX - Fragments

- En JSX l'utilisation d'une balise se traduit par un React.createElement
- On ne peut dont pas associer plusieurs balises JSX à une seule expression, par exemple

```
// Erreur (2 éléments)
const els = <div>A</div><div>B</div>;

// Erreur (2 éléments)
return <div>A</div><div>B</div>;
```

```
// Erreur (2 éléments)
<div>{<div>A</div><div>B</div>}</div>

// Erreur (2 éléments)
<div>{condition && <div>A</div><div>B</div>}</div>
```

JSX - Fragments

- Pour éviter ça on peut :
  - utiliser un tableau mais cela nous afficherait un warning si on utilise pas la prop key
  - utiliser une balise JSX supplémentaire
  - utiliser un fragment (de préférence avec la syntaxe courte)

```jsx
// Avec un tableau (il faudra utiliser key)
const el = <dl>{condition && [<dt key="k1">Term</dt>, <dd key="k2">Definition</dd>]}</dl>;

// Avec une balise supplémentaire (n'aurait pas de sens en HTML dans cet exemple)
const el = <dl>{condition && <div><dt>Term</dt><dd>Definition</dd></div>}</dl>;

// Avec un Fragment (il faut importer Fragment de react)
const el = <dl>{condition && <Fragment><dt>Term</dt><dd>Definition</dd></Fragment>}</dl>;

// Avec un Fragment en syntaxe courte (recommandé)
const el = <dl>{condition && <><dt>Term</dt><dd>Definition</dd></>}</dl>;
```

formation.tech

Composants

Composants - Introduction

- Un composant est une fonction React qui retourne ReactNode (Element, string, null...)

```jsx
function Hello() {
  return <div className="Hello">Hello, world !</div>;
}

export default Hello;
```

- Et qu'on utilisera en JSX dans un autre composant

```jsx
import Hello from './Hello';

function App() {
  return (
    <div className="App">
      <Hello />
    </div>
  );
}

export default App;
```

Composants - Règles et conventions

- En JSX composant doit obligatoirement commencer par une majuscule (les minuscules seront utilisés pour les éléments HTML)
- Lorsqu'on retourne du JSX dans un composants sur plusieurs lignes on ajoute des parenthèses pour pouvoir indenter le code :

```
function App() {
  return (
    <div className="App">
      <Hello />
    </div>
  );
}
```

- Sans les parenthèses le JSX ne serait jamais retourné car la fin de la ligne du return serait vue comme un point-virgule

```
function App() {
  return   // équivalent à return;
    <div className="App">
      <Hello />
    </div>
  ;
}
```

Composants - Règles et conventions

- Pour repérer rapidement les composants dans le DOM, ce peut être une bonne pratique d'utiliser le nom du composant en classe de la balise racine

```
▼<body>
  ▼<div id="root">
    ▼<div class="App">
       <div class="Hello">Hello, world !</div>
     </div>
```

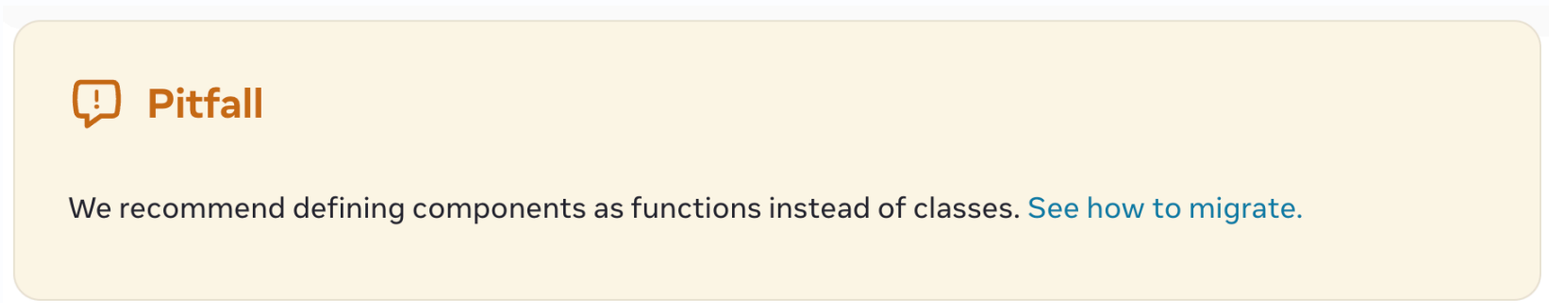
- Un composant est de préférence exporté par défaut pour pouvoir utiliser facilement React.lazy

Composants - Classes

- Historiquement un composant React pouvait également être déclaré sous forme de classe

```
class Hello extends Component {
  render() {
    return <div className="Hello">Hello, world !</div>;
  }
}
```

- La doc officielle recommande de ne plus les utiliser (sauf dans de rares cas comme les Error Boundaries)



- Avant l'arrivée des hooks (useState, useEffect...), les classes étaient le seul moyen de d'utiliser le state où les fonctions qui se déclenche sur le cycle de vie du composant (Lifecycle Hooks / Effects)
- On parlait parfois de Stateful Components en parlant des classes et Stateless Components en parlant des fonctions, aujourd'hui on dit plutôt class components ou fonction components

Composants - TypeScript

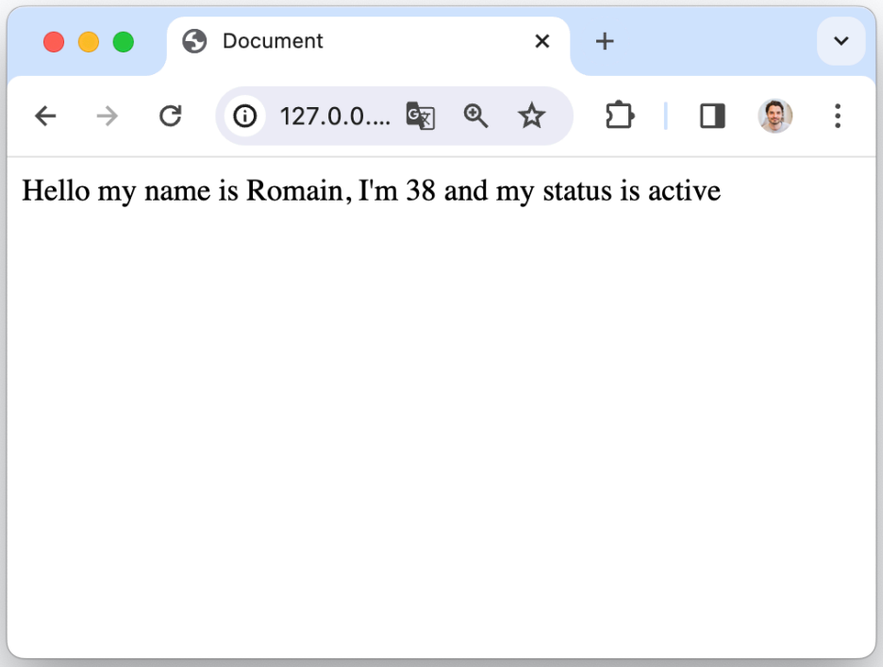
- En TypeScript si on souhaite typer explicitement le retour on pourra utiliser ReactNode :

```tsx
import { ReactNode } from 'react';

function Hello(): ReactNode {
  return <div className="Hello">Hello, world !</div>;
}
```

Composants - Props

- Les props sont les paramètres d'entrées des composants
- Elles rendent les composants réutilisables et permettent de communiquer avec le composant parent (celui qui utilise le composant enfant dans son JSX)

```jsx
function Hello(props) {
  return (
    <div className="Hello">
      Hello my name is {props.name}, I'm {props.age} and my
      status is {props.active ? 'active' : 'inactive'}
    </div>
  );
}


function App() {
  return (
    <div className="App">
      <Hello name="Romain" age={38} active />
    </div>
  );
}
```



Composants - Props

- Props est un objet, c'est le 2e paramètres de React.createElement
- Pour rappel le JSX :

```jsx
<Hello name="Romain" age={38} isActive />
```

- Devient en JS :

```jsx
React.createElement(Hello, {
  name: 'Romain',
  age: 38,
  isActive: true,
})
```

Composants - Props et complétion

- En JavaScript il n'y a pas de complétion concernant les props (les paramètres d'entrées n'étant pas typés)

```
function App() {
  return (
    <div className="App">
      <Hello />
    </div>
  );
}
```

```
abc App
abc as
abc className
abc client
```

- Afin d'améliorer la complétion on peut :
  - déstructurer props
  - utiliser JSDoc
  - utiliser prop-types
  - utiliser TypeScript

Composants - Typer props en déstructurant

- Pour récupérer rapidement des suggestions on suggère de déstructurer l'objet props dans les parenthèses de la fonction composant : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

```
function Hello({ name, age, isActive }) {
  return (
    <div className="Hello">
      Hello my name is {name}, I'm {age} and my
      status is {isActive ? 'active' : 'inactive'}
    </div>
  );
}
```

- 2 avantages
  - le JSX est plus court est donc plus lisible (name au lieu de props.name)
  - à l'utilisation en JSX les props sont suggérées à la complétion

```
function App() {
  return (
    <div className="App">
      <Hello />
    </div>
  );
}
```

```
age                    (property) age: any
isActive
name
key?
```

Composants - Typer props en déstructurant

- En passant des valeurs par défaut on peut également indiquer le type

```
function Hello({ name = '', age = 0, isActive = false }) {
  return (
    <div className="Hello">
      Hello my name is {name}, I'm {age} and my
      status is {isActive ? 'active' : 'inactive'}
    </div>
  );
}
```

- Les complétions seront encore améliorées

```
function Hello({ name = '', age = 0, isActive = false }) {
  return (
    <div className="Hello">
      Hello my name is {name.}, I'm {age} and my
      status is {isActive ? '
    </div>
  );
}
```

charAt
charCodeAt
codePointAt
concat
endsWith

Composants - Typer props avec JSDoc

- Avec JSDoc on utilise des commentaires qui commencent par 2 étoiles :
  `/**  */`

```
/**
 * Props of Hello component
 * @typedef {Object} HelloProps
 * @property {string} name - name of the user
 * @property {number} age - age of the user
 * @property {boolean} isActive - indicates whether user is active.
 */
```

```
/**
 * @type Props
 * @param {HelloProps} props
 * @returns {import("react").ReactNode}
 */
function Hello(props) {
  return (
    <div className="Hello">
      Hello my name is {props.name}, I'm {props.age} and my
      status is {props.isActive ? 'active' : 'inactive'}
    </div>
  );
}
```

- Documentation : https://jsdoc.app/

Composants - Typer props avec JSDoc

- L'avantage par rapport à la déstructuration est qu'on peut spécifier les props obligatoires, optionnelles, les types et une description
- On peut en plus déstructurer si on souhaite raccourcir les lignes en JSX

```
 */
function Hello(props) {
  return (
    <div className="Hello">
      Hello my name is {props.name}, I'm {props.age} and my
      status is {props.? 'active' : 'inactive'}
    </div>
  );
}
```

```
function App() {
  return (
    <div className="App">
      <Hello />
    </div>
  );
}

export defaul
```

Composants - Typer props avec prop-types

- Avec la déstructuration on améliorer la complétion mais pas la détection d'erreur
- On a beau déclarer une props obligatoire, rien oblige à la passer
  - Pour rendre une prop obligatoire il faudrait utiliser lorsqu'elle n'est pas définie : `throw new Error('Message')`
- On a beau déclarer un type, rien oblige à le respecter
  - Idem pour les valeurs obligatoire, il faudrait tester le type et utiliser throw pour vérifier

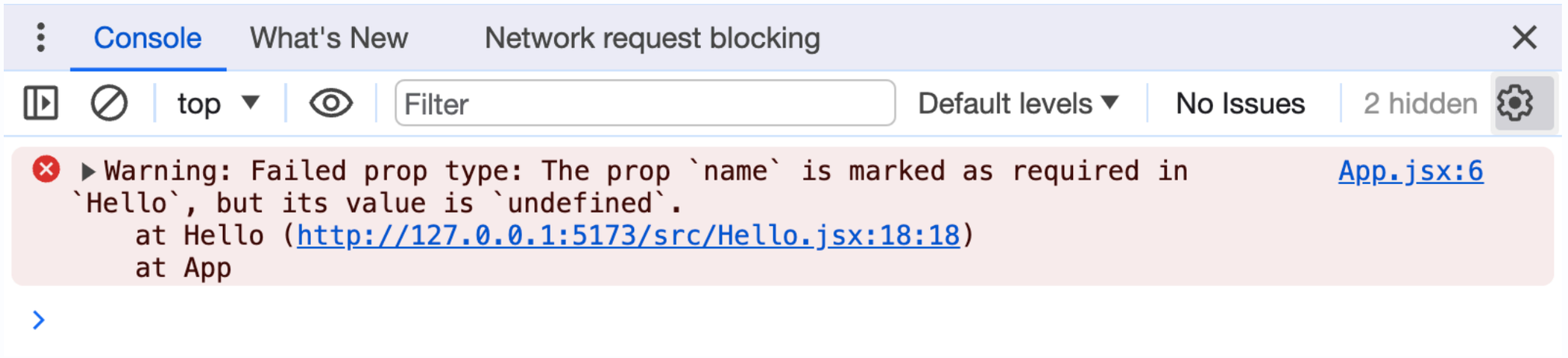
Composants - Typer props avec prop-types

- Prop Types est une bibliothèque officielle de Facebook
  - d'abord intégrée à React jusqu'à la version 15
  - proposée aujourd'hui sous forme d'un paquet séparé
- Installation :
  npm i prop-types
- Utilisation :

```
import { bool, number, string } from 'prop-types';

function Hello({ name = '', age = 0, isActive = false }) {
  return (
    <div className="Hello">
      Hello my name is {name}, I'm {age} and my
      status is {isActive ? 'active' : 'inactive'}
    </div>
  );
}

Hello.propTypes = {
  name: string.isRequired,
  age: number.isRequired,
  isActive: bool
};
```

Composants - Typer props avec prop-types

- Avec Prop Types les erreurs s'affichent à l'exécution dans la console
- Prop Types émet des warnings ce qui signifie qu'en prod ils disparaissent

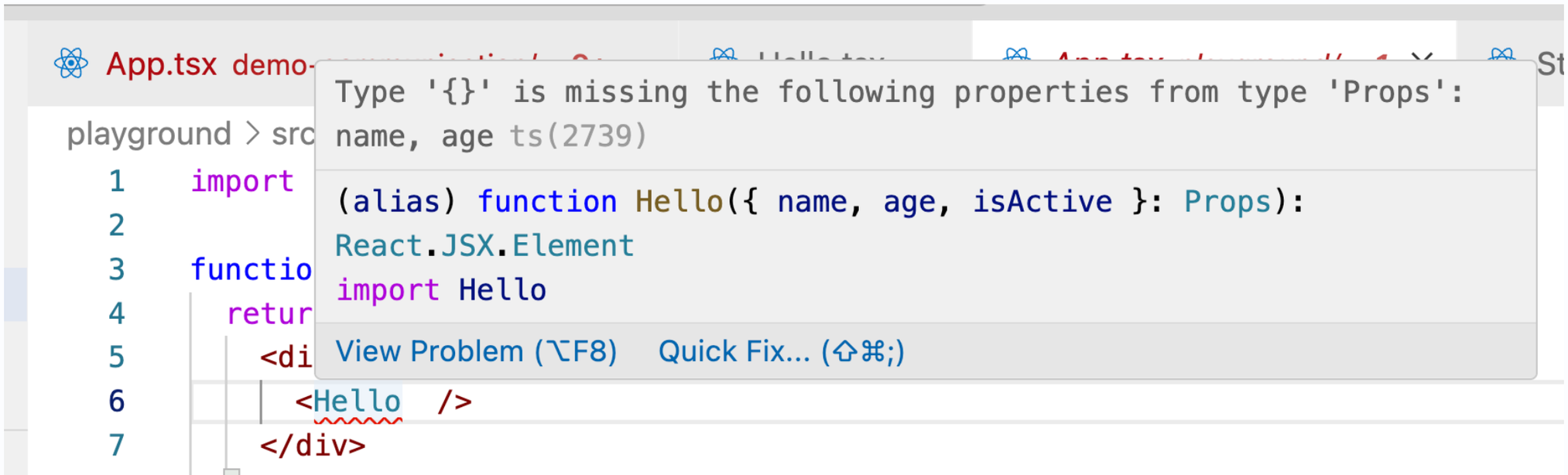


Composants - Typer props avec TypeScript

- En TypeScript on doit déclarer un object type ou une interface pour Props :

```tsx
type Props = {
  name: string;
  age: number;
  isActive?: boolean;
};

function Hello({ name = '', age = 0, isActive = false }: Props) {
  return (
    <div className="Hello">
      Hello my name is {name}, I'm {age} and my
      status is {isActive ? 'active' : 'inactive'}
    </div>
  );
}
```

- Par rapport aux autres solutions TypeScript oblige à respecter le type pour pouvoir builder et donc exécuter le code



Composants - Typer props avec TypeScript

- On peut également typer props avec une interface :

```tsx
interface Props {
  name: string;
  age: number;
  isActive?: boolean;
}

function Hello({ name = '', age = 0, isActive = false }: Props) {
  return (
    <div className="Hello">
```

- Types or Interfaces ?
  https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/basic_type_example#types-or-interfaces

  - utiliser interface si le composant à pour vocation à être partager à plusieurs projets ou via npm
  - utiliser type sinon

Composants - Pure components

- Pour éviter les bugs il faut écrire les composants React sous forme de fonctions pures
- Fonction pure :
  - prédictive, appelée avec des paramètres donnés elle aura toujours le même retour comme une formule mathématique
  - ne doit pas modifier ses paramètres d'entrées
  - ne doit pas avoir de side-effects :
    - envoyer des requêtes
    - utiliser le localStorage
    - manipuler le DOM
    - logger
    - modifier une variable de externe

Composants - Pure components

- Exemples de fonctions prédictives vs non-prédictives

```
// prédictive, sum(1, 2) === 3
function sum(a: number, b: number) {
  return a + b;
}

// non prédictive, randomInt(0, 10) === ???
function randomInt(min: number, max: number) {
  min = Math.round(min);
  max = Math.round(max);
  return Math.floor(Math.random() * max - min) + min;
}
```

- Exemples de fonctions qui modifient leurs paramètres ou non

```
// modifie ses paramètres
function addNumberMutable(array: number[], nb: number) {
  array.push(nb);
}

// ne modifie pas ses paramètres
function addNumberImmutable(array: number[], nb: number) {
  return [...array, nb];
}
```

Composants - Pure components

- Exemples de fonctions qui ont des side-effects

```
// side-effect
let count = 0;
function incrementCount() {
  count++;
}

// side-effect
async function fetchUsers() {
  const res = await fetch('https://jsonplaceholder.typicode.com/users');
  return await res.json();
}
```

- Avec React on peut garder nos composants purs avec :
  - les événements
  - useEffect
  - useState
  - useRef

formation.tech

Events

Events - Introduction

- React permet d'écouter les événements avec une syntaxe déclarative inspiré du HTML / JS des premières années

```
<button onclick="myFunction()">Click me</button>
```

- Historiquement les attributs HTML on* permettent d'écouter des événements mais obligeaient à exécuter des fonctions globales

- Avec React on utilise un API très proche (remarquez onclick vs onClick) :

```
function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}
```

- La fonction associée à l'événement est appelée Event Handler

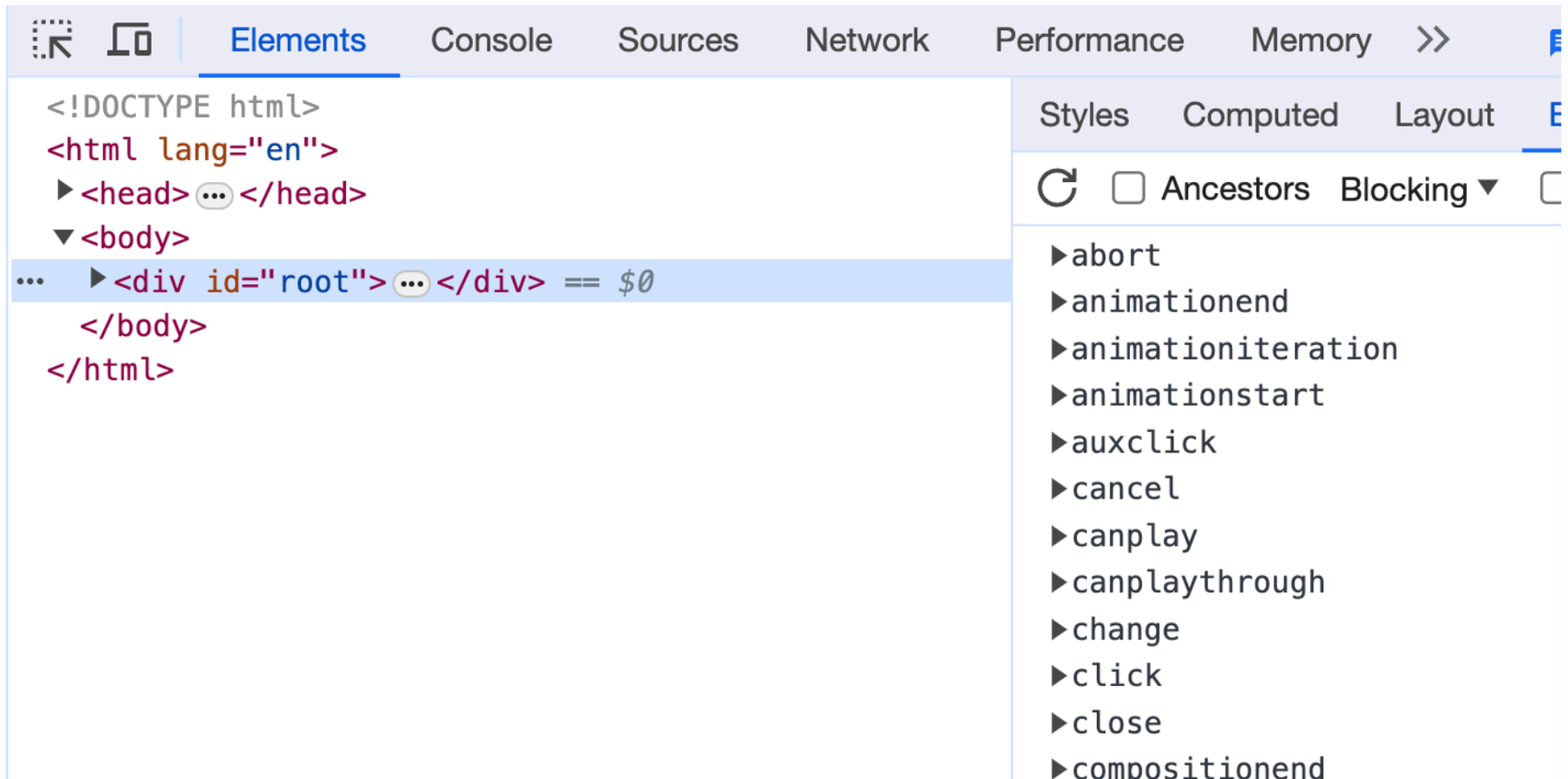
Events - Event object

- Si besoin d'interagir avec l'objet event il suffit de le récupérer dans les paramètres de l'event handler

```
function Button() {
  function handleClick(event) {
    alert(`You clicked at ${event.clientX}, ${event.clientY}`);
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}
```

Events - Event delegation

- Pour améliorer les performances de larges applications et offrir certaines fonctionnalités avancées (event replaying...), React écoute tous les événements au niveau de l'élément racine de l'application
- Les événements qui ne se propagent pas sont aliasés (focus → focusin, blur → focusout)
- Pour que le système fonctionne, React doit simuler les comportements natifs (notamment les events phases) via un objet event custom



Events - Event object

- En logguant l'événement vous remarquerez que ce n'est pas l'événement natif du navigateur :

```
function Button() {
  function handleClick(event) {
    console.log(event);
  }

  return (
    <button onClick={handleClick}>
```

```
▼ SyntheticBaseEvent {_reactName: 'onClick', _targetInst: n
nt: PointerEvent, target: button, …} i
    altKey: false
    bubbles: true
    button: 0
    buttons: 0
    cancelable: true
    clientX: 59
    clientY: 53
    ctrlKey: false
    currentTarget: null
    defaultPrevented: false
    detail: 1
    eventPhase: 3
  ▶ getModifierState: ƒ modifierStateGetter(keyArg)
  ▶ isDefaultPrevented: ƒ functionThatReturnsFalse()
  ▶ isPropagationStopped: ƒ functionThatReturnsFalse()
    isTrusted: true
    metaKey: false
    movementX: 0
    movementY: 0
  ▶ nativeEvent: PointerEvent {isTrusted: true, pointerId:
    pageX: 59
```

- L'événement React reproduit les clés le l'objet event natif
- Vous pourrez accéder au à l'event natif via la clé *nativeEvent*

Events - Event Capture

- On peut également préciser que l'évènement est écouter dans la phase de capture en ajoutant le suffixe Capture à la prop on* (ex: onClickCapture)

```
function Button() {
  function handleClick(event) {
    console.log(event);
  }

  return (
    <button onClickCapture={handleClick}>
```

- Pour rappel la phase de capture exécute les event handlers de la racine de l'arbre vers la feuille de l'arbre :
  window > document > html > body ...
- Dans la phase par défaut (phase de bubbling) les handlers s'exécutent de la feuille vers la racine
- https://javascript.info/bubbling-and-capturing

Events - onChange

- Afin de rationnaliser la gestion des événements des champs de formulaire, React propose d'écouter les événements input, change et click via un handler unique onChange

```
function UserForm() {
  function handleChange(event) {
    console.log(event.nativeEvent.type); // input, change ou click
  }

  return (
    <div>
      <input type="text" onChange={handleChange} /> {/* === onInput */}
      <input type="checkbox" onChange={handleChange} /> {/* === onClick */}
      <select onChange={handleChange}> {/* === onChange */}
          <option>Oui</option>
          <option>Non</option>
      </select>
    </div>
  );
}
```

- Les checkbox et boutons radio écoutent en réalité click
- Les balises select et input de type *file* écoutent change
- Les autres types d'input et textarea écoutent input

Events - TypeScript

- En TypeScript pour typer l'objet on devra donc importer un type React, attention souvent le type porte le même nom que le type natif (qui lui est global donc ne serait pas importé)

```tsx
import { MouseEvent } from 'react';

function Button() {
  function handleButtonClick(event: MouseEvent<HTMLButtonElement>) {
    console.log('button click', event.target);
  }
  return <button onClick={handleButtonClick}>Click me</button>;
}
```

- Les types d'événement React sont génériques (contrairement aux types natifs) et permettent de typer currentTarget (et même target pour ChangeEvent)

Events - TypeScript

- Les Event types suivants sont implémentés par React :
  ClipboardEvent, CompositionEvent, DragEvent, PointerEvent, FocusEvent, KeyboardEvent, MouseEvent, TouchEvent, UIEvent, WheelEvent, AnimationEvent, TransitionEvent
- ChangeEvent sur des éléments Input, Select ou TextArea émet un ChangeEvent
- Les autres événements de formulaire sont regroupés sous FormEvent (change sur d'autres champs, beforeinput, input, submit, reset, invalid). InputEvent et SubmitEvent n'existent pas
- Pour les autres événement on (Error event, media events, load, error events...) on utilise l'interface générique SyntheticEvent
- Les autres types d'événements ne sont pas disponibles car pas accessibles via JSX (FetchEvent, NavigationEvent, ...)

Events - TypeScript

- Si on utilise et le type natif et le type React il faudra aliaser le type React

```tsx
import { MouseEvent as ReactMouseEvent, useEffect } from 'react';

function Button() {
  useEffect(() => {
    function handleWindowClick(event: MouseEvent) {
      console.log('window click', event);
    }
    window.addEventListener('click', handleWindowClick);
    return () => {
      window.removeEventListener('click', handleWindowClick);
    };
  });

  function handleButtonClick(event: ReactMouseEvent<HTMLButtonElement>) {
    console.log('button click', event);
  }

  return <button onClick={handleButtonClick}>Click me</button>;
}
```

formation.tech

State

State - Introduction

- Comme vu en introduction, React va au moment d'un nouveau render déterminer via un diff les modifications apportées à l'application
- Au niveau d'un composant il serait compliqué de rappeler le render du fichier principal (difficulté de réutilisation)
- Faire le diff de toute l'application peut être potentiellement couteux sur de larges bases de code
- Pour provoquer le render au niveau d'un composant on utilise le state
- Dans un function component cela se fait via le hook useState
- Historiquement cela se faisait via la méthode setState de la classe Component (et obligeait donc le refactoring vers une classe)

State - Sans state

- Exemple sans state :

```
function Clock() {
  const now = new Date();

  return <div className="Clock">Il est {now.toLocaleTimeString()}</div>;
}
```

- Sans le state, provoquer un rafraîchissement de l'heure nécessiterait de rappeler le render à la racine de l'application chaque seconde :

```
const root = createRoot(document.getElementById('root') as HTMLDivElement);
root.render(<App />);

setInterval(() => {
  root.render(<App />);
}, 1000);
```

- Problèmes :
  - Clock n'est pas une fonction pure
  - root.render provoque le rafraîchissement et donc le diff de toute l'application
  - le composant est compliqué à utiliser puisqu'il faut rajouter une action à la racine

State - useState

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

function Clock() {
  const [now, setNow] = useState(new Date());

  setTimeout(() => {
    setNow(new Date());
  }, 1000);

  return <div className="Clock">Il est {now.toLocaleTimeString()}</div>;
}
```

- useState est un hook : une fonction de React qui commencent par use et ne peut être appelée que dans un function component
- useState retourne un tableau (qu'on a ici déstructuré) :
  - le premier élément est la valeur actuelle du state
  - le 2e élément est un fonction pour la mise à jour
- Le paramètre d'entrée de useState est la valeur initiale du state

State - useState

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

function Clock() {
  const [now, setNow] = useState(new Date());

  setTimeout(() => {
    setNow(new Date());
  }, 1000);

  return <div className="Clock">Il est {now.toLocaleTimeString()}</div>;
}
```

- useState aide à créer des fonctions pures (mais cet exemple ne l'est pas à cause de setTimeout, il faudrait utiliser un effet), car un prochain appel à ce composant afficherait à nouveau la valeur précédente (sauf si on appelle la fonction de mise à jour, ici setNow)
- La fonction de mise à jour (ici setNow), va provoquer le render au niveau de ce composant (et donc de ses enfants), pas de l'ensemble de l'application
- Comme Clock est rappelé on ne pourrait pas utiliser setInterval à la racine du composant (il faudra utiliser un effet)

State - useState

```jsx
function Counter() {
  const [count, setCount] = useState(0);
  const [step, setStep] = useState(1);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div className="Counter">
      <button onClick={handleClick}>{count}</button>
      <input value={step} onChange={(event) => setStep(event.target.valueAsNumber)} />
    </div>
  )
}
```

- L'ordre des useState doit toujours être le même entre 2 render
- C'est ce qui permet à React de savoir que tel useState correspond à telle valeur
- Attention à ne pas :
  - utiliser de conditions autour de useState
  - ne pas utiliser return de manière conditionnelle avant les useState
  - ne pas utiliser de boucles autour de useState

State - useState

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount((c) => c + 1);
  }

  return <button>{count}</button>
}
```

- La fonction set, peut également prendre un callback en paramètre pour mettre à jour la valeur

State - TypeScript

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

function Clock() {
  const [now, setNow] = useState(new Date());

  setTimeout(() => {
    setNow(new Date());
  }, 1000);

  return <div className="Clock">Il est {now.toLocaleTimeString()}</div>;
}
```

- La plupart du temps, useState n'a pas besoin qu'on précise le type grâce à l'inférence de type qui va le déterminer à partir de la valeur par défaut
- Parfois la valeur initiale n'est pas suffisamment précise (tableau vide, objet avec des clés optionnelles...)

```tsx
function List() {
  const [names, setNames] = useState([]);

  return (
    <div className="List">
      {names.map((name) => (
        <div key={name}>{name.toUpperCase()}</div>
      ))}
    </div>
```

formation.tech

Effects

Effects - Introduction

- Nous avons vu précédemment qu'une fonction React doit être pure
- Cela implique de ne pas appeler des API Web depuis les composants
- Pour garder ses composants purs, on utilise un effet avec useEffect

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

function Clock() {
  const [now, setNow] = useState(new Date());

  useEffect(() => {
    setInterval(() => {
      setNow(new Date());
    }, 1000);
  }, []);

  return <div className="Clock">Il est {now.toLocaleTimeString()}</div>;
}

export default Clock;
```

Effects - 2e param

- Le 2e paramètre détermine quand l'effet doit être rappelé

```
// si pas de 2e param, le callback est rappelé à chaque appel
// du composant
useEffect(() => {

});

// si un tableau vide, le callback est appelé une seule fois
// après le premier affichage du composant
useEffect(() => {

}, []);

// si un tableau remplit, le callback est appelé à chaque fois
// que la valeur (ou les valeurs) changent, une fois le rendu du composant
useEffect(() => {

}, [val1, val2]);
```

Effects - 2e param

- Si on retourne une fonction, celle-ci sera appelée pour nettoyer le composant (cleanup function)

```
// si pas de 2e param, le callback est rappelé à chaque appel
// du composant
useEffect(() => {
  return () => {
    // à chaque fois que le composant est rappelé (avant qu'il soit rappelé)
  };
});
```

```
// si un tableau vide, le callback est appelé une seule fois
// après le premier affichage du composant
useEffect(() => {
  // quand le composant disparait (destructeur)
}, []);

// si un tableau remplit, le callback est appelé à chaque fois
// que la valeur (ou les valeurs) changent, une fois le rendu du composant
useEffect(() => {
  // quand les valeurs changent (avant qu'il soit rappelé)
}, [val1, val2]);
```

Effects - 2e param

- Comme la signature d'un effet est de retourner une fonction, il n'est pas possible d'utiliser des fonctions asynchrones

```
// ERREUR
useEffect(async () => { // retourne un objet Promise

}, []);
```

- A la place on peut utiliser une IIAFE (Immediately Invoked Async Function Expression)

```
useEffect(() => {
  // IIAFE (Immediately Invoked Async Function Expression)
  (async () => {

  })();
}, []);
```

formation.tech

Communication inter-composants

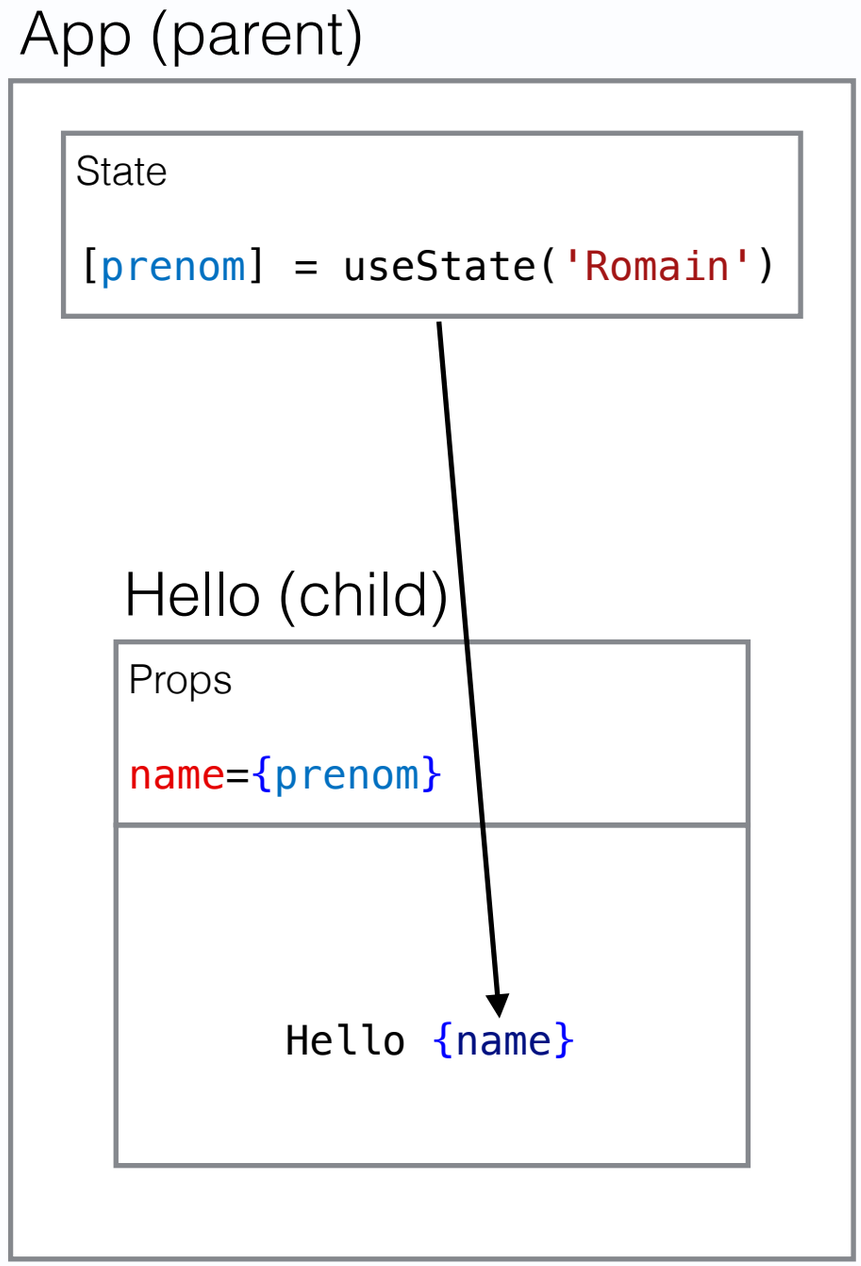
Communication - Introduction

- Pour partager une valeur entre plusieurs composants on peut :
  - utiliser un state dans l'ancêtre commun le plus proche (closest common ancestor)
  - utiliser le context
  - utiliser un store comme Redux et une intégration comme React Redux (qui va simplifier le context)

Communication - Parent vers enfant

- Pour passer une valeur d'un composant parent vers un composant enfant on utilise une prop (string, number, boolean, array, object...)

```tsx
function App() {
  const [prenom] = useState('Romain');
  return (
    <div className="App">
      <Hello name={prenom} />
    </div>
  );
}

type HelloProps = {
  name: string;
}

function Hello({ name }: HelloProps) {
  return (
    <div className="Hello">
      Hello {name}
    </div>
  )
}
```



Communication - Parent vers enfant

- Pour passer une valeur d'un composant enfant vers un composant parent, le parent doit passer une fonction en prop qui sera appelée par l'enfant

```tsx
function App() {
  const [prenom, setPrenom] = useState("Romain");
  return (
    <div className="App">
      <Hello name={prenom} onNameChange={setPrenom} />
    </div>
  );
}

type HelloProps = {
  name: string;
  onNameChange(value: string): void;
};

function Hello({ name, onNameChange }: HelloProps) {
  return (
    <div className="Hello">
      Hello {name}
      <input
        value={name}
        onChange={(e) => onNameChange(e.target.value)}
      />
    </div>
  );
}
```