formation.tech

React

formation.tech

Présentations

Présentations - Formateur

- Romain Bohdanowicz
  Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle

- Expérience
  Formateur/Développeur Freelance depuis 2006

- Langages
  Expert : HTML / CSS / JavaScript / TypeScript / PHP / Java
  Notions : C / C++ / Objective-C / C# / Python / Bash / Batch

- Certifications
  PHP / Zend Framework / Node.js

- A propos
  Premier site web à 12 ans (HTML/JS/PHP)
  Triathlète du dimanche

Présentations - Horaires

- Matin
  - 9h - 10h
  - 10h15 - 11h15
  - 11h30 - 12h30
- Après-midi
  - 13h45 - 14h45
  - 15h - 16h
  - 16h15 - 17h15

Introduction - formation.tech

- Organisme de formation référencé depuis 2016
- Certifié Qualiopi
- Environs 65 formations au catalogue en 2024
- Une dizaine de formateurs indépendants
- Formations en français ou anglais
- https://formation.tech/

Introduction - Et vous ?

- Pré-requis ?
- Rôle dans votre société ?
- Intérêt / objectif de cette formation ?

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

```html
<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/

```
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}
    className="basic-multi-select"
    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`

```
Bureau — npx create-vite — npx — node ◂ npm exec create-vite __CFBundleIdentifier=com.apple.Terminal TMPDIR=/var/fol...
➜  Desktop npx create-vite
Need to install the following packages:
  create-vite@4.4.1
Ok to proceed? (y) y
✔ Project name: … f
✔ Select a framework: › React
? Select a variant: › - Use arrow-keys. Return to submit.
❯   TypeScript
    TypeScript + SWC
    JavaScript
    JavaScript + SWC
```

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 :

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

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

- En TypeScript

```
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 :

```
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 :

```
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 :

```
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

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

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

// 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>`

```
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

```
function App() {
  let element;

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

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

```
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

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

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

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

```
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 (pour un tableau donné, pas pour toute l'app)
- 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',
  'Edouard',
]
```

→ `(name) => <div key="name">{name}</div>` →

```
[
  <div key="name">Romain</div>,
  <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>
  );
}
```

formation.tech

Composants

Composants - Introduction

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

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

export default Hello;
```

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

```
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)

> **Pitfall**
>
> We recommend defining components as functions instead of classes. See how to migrate.

- 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 :

```
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)

```
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 :

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

- Devient en JS :

```
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>
  );
}
```

age
isActive
name
Hello

(property) age: number

- age of the user

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

export defaul
```

age
isActive
name
key?
App

(property) age: number

- age of the user

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 :

```
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 :

```
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: r
  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)

```
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

```
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

```
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

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

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

  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

```
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 - useState

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

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

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

- Attention, lorsqu'on met à jour le state

State - TypeScript

```
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...)

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

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

State - Arrays / Objects

- Lorsque le state contient un objet (ou un tableau) il faut passer une copie de l'objet

State - Arrays / Objects

- Il n'est pas nécessaire et même contre-performant de cloner tous les objets
- Exemple pour modifier le nom Caroline on doit cloner : le tableau, l'objet qui contient user et address, puis l'objet qui contient name

```
[
  {
    user: {
      name: 'Romain',
    },
    address: {
      city: 'Paris'
    }
  },
  {
    user: {
      name: 'Jean',
    },
    address: {
      city: 'Bordeaux'
    }
  },
  {
    user: {
      name: 'Benjamin',
    },
    address: {
      city: 'Miami'
    }
  }
]
```

→

```
[
  {
    user: {
      name: 'Romain',
    },
    address: {
      city: 'Paris'
    }
  },
  {
    user: {
      name: 'Caroline',
    },
    address: {
      city: 'Bordeaux'
    }
  },
  {
    user: {
      name: 'Benjamin',
    },
    address: {
      city: 'Miami'
    }
  }
]
```

State - Arrays / Objects

- En terme de code :

```
setValues([
  ...values.slice(0, i),
  {
    ...values[i],
    user: {
      ...values[i].user,
      name: newName
    }
  },
  ...values.slice(i + 1),
]);
```

- Le code est particulièrement illisible

State - Arrays / Objects

- Pour améliorer la lisibilité on peut utiliser une bibliothèque comme Immer qui va traduire du code muable en version immuable : https://immerjs.github.io/immer/

```
import { produce } from "immer";

const nextValues = produce(values, (draft) => {
  draft[1].user.name = newName;
});

setValues(nextValues);
```

- Directement avec le custom hook useImmer

```
import { useImmer } from "use-immer";

const [values, setValues] = useImmer([
  { user: { name: 'Romain' }, address: { city: 'Paris' } },
  { user: { name: 'Caroline' }, address: { city: 'Bordeaux' } },
  { user: { name: 'Benjamin' }, address: { city: 'Miami' } },
]);

setValues((draft) => {
  draft[1].user.name = newName;
});
```

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(() => {
  return () => {
     // 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(() => {
  return () => {
     // 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...)

```
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

```
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>
  );
}
```

Communication - Via les props

- Pour communiquer via les props on doit trouver l'ancêtre commun le plus proche, inconvénients :
  - lorsque les composants sont éloignés il faut parfois traverser plusieurs composants qui ne sont pas concernés par le changement
  - modifier une props implique de re-rendre à nouveau tous les composants
  - https://github.com/formation-tech/react-communication/tree/master/src/example-props

Communication - Via le context

- Via le context
  - le context permet une communication en traversant la hiérachie, un state défini à un niveau n, peut être directement utilisé à un niveau n+2, n+3...
  - l'API est historiquement compliqué à utiliser, de nombreuses personnes ont utilisé des stores comme Redux pour simplifier son utilisation
  - https://github.com/formation-tech/react-communication/tree/master/src/example-context-with-hooks

Communication - Via le context

- Apparu en 2013, documenté à partir de 2015 (API déprécié depuis)
- L'API actuel simplifie son utilisation (depuis 2018)

formation.tech

SSR

SSR - Introduction

- Le Server Side Rendering (SSR ou Rendu côté serveur) est une technique qui consiste à exécuter le code React côté serveur afin que le HTML généré par React soit présent dans la réponse HTTP
- 3 avantages :
  - amélioration du SEO (Search Engine Optimization)
  - aperçu lors du partage sur les réseaux sociaux
  - performance car le traitement effectué côté serveur peut être mis en cache
- Nécessite une plateforme qui supporte le JavaScript comme Node.js ou Deno
- Pour utiliser le SSR il est fortement recommandé de passer par un framework : Next.js ou Remix

SSR - APIs serveur

- Pour mettre en place le SSR sans framework on utilise le paquet react-dom/server
- https://react.dev/reference/react-dom/server
- Ce paquet contient 5 fonctions :
  - renderToString
  - renderToStaticMarkup
  - renderToPipeableStream
  - renderToStaticNodeStream
  - renderToReadableStream (pour les environnements qui supportent les Web Streams comme Deno)
- Pour améliorer les performances il est conseillé d'utiliser les streams

SSR - APIs serveur

- La mise en place est plus complexe avec les streams mais plus performante (le rendu peut commencer même si les appels asynchrones ne sont pas terminés)
- renderToStaticMarkup et renderToStaticNodeStream sont pour des sites web "statiques", c'est à dire pour lesquels il n'y a pas d'interactions utilisateur (formulaires, clicks...)
- Le HTML ayant déjà été créé côté serveur, coté client on appeler hydrateRoot au lieu de createRoot au moment de démarrer l'application (react-dom/client), hydrateRoot va lier React au HTML existant (event listeners...)

SSR - Mise en place

- Avec Vite on devra builder 2 applications "client" et "serveur"

```
"build": "npm run build:client && npm run build:server",
"build:client": "tsc && vite build --outDir dist/client",
"build:server": "tsc && vite build --ssr src/ssr.tsx --outDir dist/server",
```

- Exemple de point d'entrée serveur qui utilise Express

```
import { renderToString } from 'react-dom/server';
import express from 'express';
import App from './App';
import { readFile } from 'fs/promises';

const app = express();

app.get('/', async (_, res) => {
  const indexHtml = await readFile('dist/client/index.html', { encoding: 'utf-8' });
  const reactHtml = renderToString(<App />);
  res.send(indexHtml.replace('<div id="root">', '<div id="root">' + reactHtml));
});

app.use(express.static('dist/client'));


app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

SSR - Mise en place

- Exemple de repos qui mettent en place React avec Vite et le SSR :
  https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-react-ts
  https://github.com/letientai299/vite-react-ssr-typescript
- Si on utilise un router comme react-router-dom il faudra également suivre les instructions :
  https://reactrouter.com/en/main/guides/ssr
- Une fois un exemple fonctionnel il faudra encore :
  - gérer les calls asynchrones (requêtes HTTP)
  - personnaliser les entêtes de certaines pages (balises title, meta...)
  - ...
- Un framework comme Next.js ou Remix simplifie tout cela

SSR - Next.js

- Next.js est un framework créé par Vercel (hébergeur cloud de projets JS)
- Parmi les 8 plus gros contributeurs React au T1 2024 (plus de 10 commits) :
  - Facebook : 4
  - Vercel : 4
- Next.js utilise en interne une version de React qui n'a pas encore été publié (probablement dans React 19)
- Next.js supporte le SSR nativement ainsi que le SSG (Static Site Generator ou générateur de site statiques, docs, b
- Création d'un projet :
  npx create-next-app@latest

```
Bureau — romain@MacBook-Pro-de-Romain — ~/Desktop — -zsh — 73×15
→  Desktop npx create-next-app@latest
Need to install the following packages:
create-next-app@14.1.4
Ok to proceed? (y) y
✔ What is your project named? … my-app
✔ Would you like to use TypeScript? … No / Yes
✔ Would you like to use ESLint? … No / Yes
✔ Would you like to use Tailwind CSS? … No / Yes
✔ Would you like to use `src/` directory? … No / Yes
✔ Would you like to use App Router? (recommended) … No / Yes
✔ Would you like to customize the default import alias (@/*)? … No / Yes
Creating a new Next.js app in /Users/romain/Desktop/my-app.

Using npm.
```

SSR - Next.js

- A la manière de Vite, Next propose des commandes pour démarrer un serveur de dev (next dev), builder (next build) et démarrer l'app en prod (next start)

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "lint": "next lint"
},
```

- Next.js est un "Convention over configuration framework" (comme Ruby on Rails, Laravel), pour créer des pages on doit placer et nommer des fichiers d'une certaine façon dans l'arborescence du projet
- La page d'index se trouve dans src/app/page.tsx (ou jsx), elle sera injecté dans le layout présent dans src/app/layout.tsx
- Documentation du routeur : https://nextjs.org/docs/app/building-your-application/routing

SSR - Next.js

- Voici les principaux fichiers qui composent une page :

- Ils peuvent s'imbriquer :

SSR - Next.js

- Les liens se font comme avec react-router-dom avec un composant Link

```
<nav>
  <Link href="/">Home</Link>
  <Link href="/users">Users</Link>
</nav>
```

- Pour gérer les métadonnées de la page (balise title, meta...) il suffit d'exporter un objet "metadata"

```
export const metadata: Metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};
```

SSR - Next.js

- Pour récupérer des données il suffit d'utiliser fetch et async/await dans son composant page

```
async function getData() {
  const res = await fetch('https://jsonplaceholder.typicode.com/users');

  if (!res.ok) {
    throw new Error('Failed to fetch data');
  }

  return res.json();
}

export default async function Users() {
  const data = await getData();

  return (
    <main>
      <h2>Users</h2>
      {data.map((user) => (
        <p key={user.id}>{user.name}</p>
      ))}
    </main>
  );
}
```

formation.tech

React Native

React Native - Introduction

- En plus de react-dom/client et react-dom/server, Facebook nous propose de faire le rendu de nos composants sous forme d'application mobile native iOS ou Android
- Contrairement à PhoneGap/Cordova qui se contente de mettre une WebView en plein écran dans une application mobile, React Native fait appel en arrière plan aux vrais composants natifs (Boutons, Menus...)
- Prérequis macOS :
  - Android : Node, Watchman, the React Native command line interface, a JDK, and Android Studio.
  - iOS : Node, Watchman, the React Native command line interface, Xcode and CocoaPods
- Prérequis Windows :
  - Android : Node, the React Native command line interface, a JDK, and Android Studio.
- Installation :
  npx react-native@latest init AwesomeProject

React Native - Exemple

```
function App(): React.JSX.Element {
  const isDarkMode = useColorScheme() === 'dark';

  return (
    <SafeAreaView>
      <ScrollView
        contentInsetAdjustmentBehavior="automatic"
      >
        <Header />
        <View
          <Section title="Step One">
            Edit App.tsx to change this
            screen and then come back to see your edits.
          </Section>
          <Section title="See Your Changes">
            <ReloadInstructions />
          </Section>
          <Section title="Debug">
            <DebugInstructions />
          </Section>
          <Section title="Learn More">
            Read the docs to discover what to do next:
          </Section>
          <LearnMoreLinks />
        </View>
      </ScrollView>
    </SafeAreaView>
  );
}
```

React Native - Metro

- Pour builder son application React Native on utilise Metro (équivalent de Vite/Webpack)

React Native - Demo

- Pour lancer l'app dans le simulateur iOS on utilise npm run ios (ou bien on appuie sur la touche i depuis metro)

formation.tech

React Avancé

React Avancé - 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>
```

React Avancé - 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)

```
// 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>;
```

React Avancé - Children

- Lorsqu'on crée un composant il est possible en JSX d'écrire directement entre la balise ouvrante et fermante, comme on le ferait en écrivant dans une balise div :

```
import { MouseEventHandler, PropsWithChildren } from "react";

type Props = PropsWithChildren<{
  onClick: MouseEventHandler;
}>;

function Button({ children, onClick }: Props) {
  return <button onClick={onClick}>{children}</button>;
}

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

- En TypeScript on peut se simplifier le typage avec PropsWithChildren (qui va typer children avec ReactNode | undefined
- Parfois children peut être une fonction (cf Render Props)

React Avancé - Ref

- Les refs permettent de récupérer une référence sur un élément du DOM comme on l'aurait fait avec document.getElementById ou document.querySelector
- Il faut attendre que le composant ait été rendu et que le DOM ait été mis à jour pour récupérer une valeur via ref.current
- Si la ref est nécessaire au chargement du composant (intégration avec des libs externes...) on doit utiliser useEffect

```
function HelloRef(): ReactNode {
  const divRef = useRef<HTMLDivElement>(null);

  useEffect(() => {
    console.log(divRef.current?.innerText); // Hello, world!
  }, []);

  return (
    <div ref={divRef} className="HelloRef">
      Hello, world!
    </div>
  );
}
```

React Avancé - Forwarding Refs

- Pour passer une ref d'un composant parent à enfant on utilise forwardRef

```
import { forwardRef } from "react";

type Props = {
  src: string;
  type: string;
  width: string | number;
};

const VideoPlayer = forwardRef<HTMLVideoElement, Props>(
  function VideoPlayer({ src, type, width }: Props, ref) {
    return (
      <video width={width} ref={ref}>
        <source src={src} type={type} />
      </video>
    );
  }
);

function App() {
  const videoRef = useRef<HTMLVideoElement>(null);

  return (
    <div className="App">
      <button onClick={() => videoRef.current?.play()}>Play</button>
      <button onClick={() => videoRef.current?.pause()}>Pause</button>
      <br />
      <VideoPlayer
        ref={videoRef}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
        type="video/mp4"
        width="250"
```

React Avancé - Forwarding Refs

- Techniquement il est également possible de passer une ref via les props mais il faudra la nommer autrement que ref
- Plutôt que d'exposer directement un élément du DOM via une ref on peut également exposer notre propre API avec useImperativeHandle

```
export interface VideoHandle {
  playFromStart(): void;
  pause(): void;
}

const VideoPlayer = forwardRef<VideoHandle, Props>(
  function VideoPlayer({ src, type, width }: Props, ref) {
    const videoRef = useRef<HTMLVideoElement>(null);

    useImperativeHandle(ref, () => ({
      playFromStart() {
        if (videoRef.current) {
          videoRef.current.currentTime = 0;
          videoRef.current.play();
        }
      },
      pause() {
        videoRef.current?.pause();
      }
    }), [])

    return (
      <video width={width} ref={videoRef}>
        <source src={src} type={type} />
      </video>
```

React Avancé - Forwarding Refs

- useImperativeHandle

```
import { useRef } from "react";
import VideoPlayer, { VideoHandle } from "./VideoPlayer";

function App() {
  const videoRef = useRef<VideoHandle>(null);

  return (
    <div className="App">
      <button onClick={() => videoRef.current?.playFromStart()}>Play from start</button>
      <button onClick={() => videoRef.current?.pause()}>Pause</button>
      <br />
      <VideoPlayer
        ref={videoRef}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
        type="video/mp4"
        width="250"
      />
    </div>
  );
}
```

React Avancé - Render Props

- Une render props est une fonction passée en props qui va faire un rendu à l'intérieur d'un composant enfant
- Un peu à la manière d'un props qui remonte une valeur au parent, la render prop va remonter une valeur pour que le parent décide du rendu
- C'est nécessaire si le parent n'a pas déjà accès à la valeur, en général lorsque le composant enfant va boucler dans son rendu, cela permet au parent de décider du render de chaque élément de la boucle

```
type Props = {
  items: string[];
  renderItem(item: string): ReactNode;
};

function Select({ items, renderItem }: Props): ReactNode {
  return (
    <div className="Select">
      <div className="menu">{items.map((item) => renderItem(item))}</div>
    </div>
  );
}

function App() {
  return (
    <div className="App">
      <Select items={['Rouge', 'Vert', 'Bleu']} renderItem={(item) => <div>{item}</div>} />
    </div>
  );
```

React Avancé - Render Props

- Render Props était également utilisé historiquement pour enrichir un composant en combinant avec props.children, qui devenait alors une fonction :

```
import { Translation } from 'react-i18next';

export function MyComponent() {
  return (
    <Translation>
      {
        (t, { i18n }) => <p>{t('my translated text')}</p>
      }
    </Translation>
  )
}
```

- Aujourd'hui on parvient au même résultat beaucoup plus facilement avec un custom hook :

```
import { useTranslation } from 'react-i18next';

export function MyComponent() {
  const { t, i18n } = useTranslation();

  return <p>{t('my translated text')}</p>
}
```

React Avancé - Higher Order Components

- Un higher order component est une fonction qui prend un composant en entrée et retourne ce composant encapsulé dans un composant qui l'enrichi :
- `const EnhanceComponent = enhance(Component)`
- Aujourd'hui on privilégiera plutôt les custom hooks pour accéder à des valeurs
- Exemple de HOC extrait de react-i18next :

```
import { withTranslation } from 'react-i18next';

function MyComponent({ t, i18n }) {
  return <p>{t('my translated text')}</p>
}

export default withTranslation()(MyComponent);
```

- Même exemple avec un custom hook :

```
import { useTranslation } from 'react-i18next';

export function MyComponent() {
  const { t, i18n } = useTranslation();

  return <p>{t('my translated text')}</p>
}
```

React Avancé - Portals

- Certain composant doivent s'afficher avec la propriété CSS position: absolute
- La position dépendrait alors de la où ils sont utilisé sur la page, pour éviter les bugs on les affiche à la racine de body
- Exemple : modal, popover...

```
<body>
  <div id="root"></div>
  <div id="modal"></div> <!-- <-- La modal doit s'afficher ici -->
</body>
```

```
function App() {
  const [showModal, setShowModal] = useState(false);
  return (
    <>
      <button onClick={() => setShowModal(true)}>
        Show modal using a portal
      </button>
      {showModal &&
        createPortal(
          <Modal>Modal content</Modal>,
          document.getElementById("modal") as HTMLElement
        )}
    </>
  );
}
```

```
▶ <div id="root">…</div>
▼ <div id="modal">  == $0
    <div class="Modal">Modal content</div>
  </div>
```

React Avancé - Error Boundaries

- Lorsqu'un composant lance une erreur lors d'un render, on peut l'intercepter avec un Error Boundary qui fonctionne comme un try .. catch
- Comme try .. catch on ne peut intercepter que les erreurs synchrone (pas celles associées à des événements ou des requêtes HTTP par exemple)
- Exemple de composant qui va crasher :

```
import { ReactNode, useState } from "react";

function CrashOnClick(): ReactNode {
  const [error, setError] = useState("");

  if (error) {
    throw new Error("CrashOnClick");
  }

  return (
    <div className="CrashOnClick" onClick={() => setError("Crash in CrashOnClick component")}>
      Will crash on click
    </div>
  );
}

export default CrashOnClick;
```

React Avancé - Error Boundaries

- Pour intercepter l'erreur on utilise un Error Boundary, malheureusement l'API n'est disponible que sous forme de classe :

```
class MyErrorBoundary extends Component<Props, State> {
  state: Readonly<State> = {
    errorMsg: "",
  };
  static getDerivedStateFromError(error: Error): State {
    return {
      errorMsg: error.message,
    };
  }
  componentDidCatch(error: Error, errorInfo: ErrorInfo): void {
    console.log("TODO send to server", error, errorInfo);
  }
  render(): ReactNode {
    const { children } = this.props;
    const { errorMsg } = this.state;

    if (errorMsg) {
      return (
        <>
          This error occured : {errorMsg}
          <button onClick={() => this.setState({ errorMsg: "" })}>Recover</button>
        </>
      );
    } else {
      return children;
    }
  }
}
```

React Avancé - Error Boundaries

- A l'utilisation l'error boundary doit englober les composants dont il interceptera les erreurs :

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

React Avancé - Error Boundaries

- Pour ne pas avoir à utiliser les classes on peut utiliser la bibliothèque react-error-boundary :

```
import { ErrorBoundary, FallbackProps } from "react-error-boundary";

function fallbackRender({ error, resetErrorBoundary }: FallbackProps) {
  return (
    <>
      This error occured : {error.message}
      <button onClick={() => resetErrorBoundary()}>Recover</button>
    </>
  );
}

function App() {
  return (
    <div className="App">
      <ErrorBoundary
        fallbackRender={fallbackRender}
        onReset={(details) => {
          // Reset the state of your app so the error doesn't happen again
        }}
      >
        <CrashOnClick />
      </ErrorBoundary>
    </div>
  );
}
```

React Avancé - Custom Hooks

- Un custom hook est une fonction qui va appeler des hooks de base comme useState, useRef, useEffect...
- Cette fonction respecter les mêmes règles que les autres Hooks :
  - commencer par use
  - n'être appelée que par des function components React ou d'autres custom hooks
  - être toujours appelée dans le même ordre (pas de if, for, return/throw avant)

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

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

  return now;
}
```

```
function Clock(): ReactNode {
  const now = useNow();
  return (
    <div className="Clock">
      {now.toLocaleTimeString()}
    </div>
  );
}
```

formation.tech

Forms

Forms - Introduction

- Avec React on a plusieurs possibilités pour récupérer les données d'un formulaire :
  - A la saisie : de façon "contrôlée" (controlled) avec un state
  - Au submit : avec l'objet event au submit
  - Au choix (saisie, blur, submit) : de façon "non-contrôlée" (uncontrolled) avec une ref

Forms - Controlled

- Un composant est dit contrôlé lorsque la valeur qui lui ai passée provient du state du composant parent
- Cela vaut aussi bien pour les champs de formulaire HTML (input, select, textarea) qui sont des composants du point de vue de React, que pour des composants React créés de toute pièce (exemple : React Select)
- Lorsque le champs est contrôlé, React a connaissance de l'état du formulaire et peut déterminer les erreurs à la saisie
- Cependant à chaque fois qu'on va saisir dans un champ, on va modifier le state et donc re-rendre le composant parent ainsi que ses descendant (peut poser des problèmes de performance)

Forms - Controlled

- Exemple avec un input :

```
function Hello() {
  const [name, setName] = useState('Bond');

  return (
    <div className="Hello">
      Name : <input value={name} onChange={(e) => setName(e.target.value)} />
      <p>The name is {name}</p>
    </div>
  );
}
```

- Exemple avec React Select :

```
function Hello() {
  const names = ["Bond", "Dr No", "Blofeld"];
  const [name, setName] = useState("Bond");

  return (
    <div className="Hello">
      Name :
      <Select
        options={names.map((name) => ({ value: name, label: name }))}
        value={{ value: name, label: name }}
        onChange={(option) => setName(option?.value ?? "")}
      />
      <p>The name is {name}</p>
    </div>
  );
}
```

Forms - Controlled

- Exemple d'un formulaire avec plusieurs champs associés à un seul state :

```
function UserFormControlled(): ReactNode {
  const [user, setUser] = useState<User>({
    username: 'romain',
    active: true,
    description: 'Bonjour...',
    gender: 'Female'
  });

  function handleChange(event: ChangeEvent<HTMLInputElement & HTMLTextAreaElement & HTMLSelectElement>) {
    const value = event.target.type === 'checkbox' ? event.target.checked : event.target.value;
    setUser({...user, [event.target.name]: value})
  }

  return (
    <form className="UserFormControlled">
      <div>
        Username :
        <input type="text" name="username" value={user.username} onChange={handleChange} />
      </div>
      <div>
        Active :
        <input type="checkbox" name="active" checked={user.active} onChange={handleChange} />
      </div>
      <div>
        Description :
        <textarea name="description" value={user.description} onChange={handleChange}></textarea>
      </div>
      <div>
        Gender :{' '}
        <select name="gender" value={user.gender} onChange={handleChange}>
          <option>Male</option>
```

Forms - Uncontrolled

- Avec des ref, de façon "non-contrôlée" :
- L'affichage des erreurs peut se faire lorsqu'on le souhaite (à la saisie, au blur, au submit) :

```
function UserFormUncontrolled(): ReactNode {
  const usernameRef = useRef<HTMLInputElement>(null);
  const passwordRef = useRef<HTMLInputElement>(null);

  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();

    const user: User = {
      username: usernameRef.current?.value,
      password: passwordRef.current?.value,
    };

    console.log(user);
  }

  return (
    <form className="UserFormUncontrolled" onSubmit={handleSubmit}>
      <div>
        Username : <input ref={usernameRef} type="text" name="username" />
      </div>
      <div>
        Password : <input ref={passwordRef} type="password" name="password" />
      </div>
      <div>
        <button>Submit</button>
      </div>
```

Forms - Uncontrolled

- On peut également récupérer les données du formulaire sans state et sans ref au submit en utilisant l'objet Event :

```
function UserFormSSR(): ReactNode {
  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();

    const data = new FormData(event.currentTarget);
    console.log(Object.fromEntries(data.entries()));
  }

  return (
    <form className="UserFormSSR" onSubmit={handleSubmit}>
      <div>
        Username : <input type="text" name="username" />
      </div>
      <div>
        Password : <input type="password" name="password" />
      </div>
      <div>
        Active : <input type="checkbox" name="active" />
      </div>
      <div>
        Description : <textarea name="description"></textarea>
      </div>
      <div>
        Gender :{' '}
        <select name="gender">
          <option>Male</option>
          <option>Female</option>
          <option>Other</option>
        </select>
```

Forms - Bibliothèques

- Il existe de nombreuses bibliothèques pour gérer les formulaires :

Forms - React Hook Form

- Pour simplifier la gestion des formulaires on peut utiliser des bibliothèques
- Plusieurs ont été populaires : Redux Form, Formik et maintenant React Hook Form
- React Hook Form propose :
  - le choix sur la gestion des formulaires : contrôlés ou non-contrôlés
  - la validation native où via des libs dédiées : Yup, Zod, Joi, Superstruct ou custom
  - la validation automatique au : change, blur, submit, touched (après le premier blur uniquement)
  - l'intégration avec les bibliothèques d'UI et les stores

Forms - React Hook Form

- Pourquoi utiliser react-hook-form ?

	React Hook Form	Formik	Redux Form
Component	uncontrolled & controlled	controlled	controlled
Rendering	minimum re-render and optimise computation	re-render according to local state changes (As you type in the input.)	re-render according to state management lib (Redux) changes (As you type in the input.)
API	Hooks	Component (RenderProps, Form, Field) + Hooks	Component (RenderProps, Form, Field)
Package size	Small `react-hook-form@7.27.0` **`8.5KB`**	Medium `formik@2.1.4` **`15KB`**	Large `redux-form@8.3.6` **`26.4KB`**
Validation	Built-in, Yup, Zod, Joi, Superstruct and build your own.	Build yourself or Yup	Build yourself or Plugins
Learning curve	Low to Medium	Medium	Medium

formation.tech

Bonnes Pratiques

Bonnes Pratiques - Conventions

- Facebook ne propose pas de conventions officielles, selon l'ancienne doc c'est même volontaire (React se disait "unoptionnated")
- Trop peu de docs traitent de conventions, exemple :
  - les imports / exports https://react.dev/learn/importing-and-exporting-components#exporting-and-importing-multiple-components-from-the-same-file
  - structure des fichiers https://legacy.reactjs.org/docs/faq-structure.html
- On trouve quelques "Style Guides" sur le web mais ils sont parfois anciens
- Conseils
  - rédigez vos propres conventions dans un fichier README.md ou équivalent
  - créer vos propres Snippets / Live Templates et les versionner dans le projet

Bonnes Pratiques - Conventions

- Exemples de conventions :
  - organisation des fichiers : par fonctionnalité, par type de fichier ou un mélange des 2 ?

```
src
├── invoices
│   ├── InvoiceDetail.tsx
│   ├── Invoices.tsx
│   ├── api.ts
│   └── hooks.ts
└── users
    ├── Login.tsx
    ├── Register.tsx
    ├── api.ts
    └── hooks.ts
```

```
src
├── api
│   ├── invoices.ts
│   └── users.ts
├── components
│   ├── InvoiceDetail.tsx
│   ├── Invoices.tsx
│   ├── Login.tsx
│   └── Register.tsx
└── hooks
    ├── invoices.ts
    └── users.ts
```

  - un composant par fichier ou plusieurs composants par fichier ?
  - une balise à la racine du composant ou un Fragment ? Si balise className avec le nom du composant ?
  - export default ou export ?

Bonnes Pratiques - Thinking in React

- Dans son chapitre Thinking in React, la documentation officielle nous présente comment nous devrions découper une maquette en composants : https://react.dev/learn/thinking-in-react

-

Bonnes Pratiques - Typer

- Comme vu précédemment on peut typer avec des commentaires JSDoc, en déstructurant, avec prop-types ou un language comme Flow ou TypeScript
- En fonction du projet choisir l'approche la plus adaptée
- La tendance actuelle est d'utiliser TypeScript

Bonnes Pratiques - Snippets / Live Templates

- Dans les IDEs comme VSCode ou WebStorm/IntelliJ on peut rédiger des raccourcis de code appelés Snippets (dans VSCode), Live Templates (dans WebStorm/IntelliJ)
- Dans VSCode on trouve de nombreux plugins qui proposent des snippets déjà écrits mais ils sont en général ancien et peuvent différer de vos conventions
- Ecrivez les vôtres et versionner les dans le projet :

```
{
  // .vscode/react.code-snippets
  "Create Function Component": {
    "scope": "typescriptreact",
    "prefix": "cfc",
    "body": [
      "import { ReactNode } from \"react\"",
      "",
      "function ${1:$TM_FILENAME_BASE}(): ReactNode {",
      " return (",
      "   <div className=\"${1:$TM_FILENAME_BASE}\">",
      "      ",
      "   </div>",
      " )",
      "}",
      "",
      "export default ${1:$TM_FILENAME_BASE};",
      ""
    ]
  }
}
```

Bonnes Pratiques - StoryBook

- StoryBook est une bibliothèque qui propose de développer ses composants (qu'ils soit UX ou représentant des pages) de façon isolés (en se concentrant sur l'API props, state...)
- Une fois développé StoryBook peut être vu comme un catalogue de composant, notamment pour les composants qui ont pour vocation à être partagés facilitant la collaboration au sein de l'équipe
- Documentation
  https://storybook.js.org/

Bonnes Pratiques - StoryBook

- Via la commande : `storybook dev -p 6006`

React - Outils de debug

- React Developer Tools
  - Extension officielle de Facebook
  - Fonctionne avec Chrome et Firefox
  - Permet de surveiller les objets props, state, context
- Téléchargement
  - Chrome
    https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
  - Firefox
    https://addons.mozilla.org/fr/firefox/addon/react-devtools/

React - Outils de debug

React - Outils de debug

formation.tech

Immuabilité

Immuabilité - Introduction

- Lors de la modification d'un objet, le changement peut-être muable en modifiant l'objet d'origine ou immuable en créant un nouvel objet
- Les algorithmes de détections de changements préfèreront les changements immuables, ayant ainsi juste à comparer les références plutôt que l'ensemble du contenu de l'objet
- Exemple, en JS les tableaux sont muables, les chaines de caractères immuables

```
const firstName = 'Romain';
firstName.concat(' Edouard');
console.log(firstName); // Romain

const firstNames = ['Romain'];
firstNames.push('Edouard');
console.log(firstNames); // Romain,Edouard
```

Immuabilité - Tableaux

- Ajouter à la fin

```
const firstNames = ['Romain', 'Edouard'];

function append(array, value) {
  return [...array, value];
}

const newfirstNames = append(firstNames, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
const firstNames = ['Romain', 'Edouard'];

function prepend(array, value) {
  return [value, ...array];
}

const newfirstNames = prepend(firstNames, 'Jean');
console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Ajouter à un indice donné

```
const firstNames = ['Romain', 'Edouard'];

function insertAt(array, value, i) {
  return [
    ...array.slice(0, i),
    value,
    ...array.slice(i),
  ];
}

const newfirstNames = insertAt(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Modifier un élément

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  return [
    ...array.slice(0, i),
    value,
    ...array.slice(i + 1),
  ];
}

const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Modifier un élément (alternative)

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  const newArray = [...array];
  newArray[i] = value
  return newArray;
}

const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Modifier un élément (alternative avec .map)

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  return array.map((el, currentI) => (i === currentI ? value : el));
}

const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Supprimer un élément

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  return [
    ...array.slice(0, i),
    ...array.slice(i + 1),
  ];
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Supprimer un élément (alternative)

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  const newArray = [...array];
  newArray.slice(i, 1);
  return newArray;
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Tableaux

- Supprimer un élément (alternative avec .filter)

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  return array.filter((el, currentI) => i !== currentI);
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Objet

- Ajouter un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function add(object, key, value) {
  return {
    ...object,
    [key]: value,
  };
}

const newContact = add(contact, 'city', 'Paris');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

Immuabilité - Objet

- Modifier un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function modify(object, key, value) {
  return {
    ...object,
    [key]: value,
  };
}

const newContact = modify(contact, 'firstName', 'Thomas');
console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

Immuabilité - Objet

- Supprimer un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function remove(object, key) {
  const { [key]: val, ...rest } = object;
  return rest;
}

const newContact = remove(contact, 'lastName');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

Immuabilité - Immutable.js

- Pour simplifier la manipulation d'objets ou de tableaux immuables, Facebook a créé Immutable.js
- Installation
  `npm install immutable`

Immuabilité - Immutable.js List

- Ajouter à la fin

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

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.push('Jean');
console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

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

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.unshift('Jean');
console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js List

- Ajouter à un indice donné

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

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.insert(1, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js List

- Modifier un élément

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

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.set(1, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js List

- Supprimer un élément

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

const firstNames = immutable.List(['Romain', 'Edouard']);

const newfirstNames = firstNames.delete(1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immutable.js Map

- Ajouter un élément

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

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.set('city', 'Paris');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

Immuabilité - Immutable.js Map

- Modifier un élément

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

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.set('firstName', 'Thomas');
console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

Immuabilité - Immutable.js Map

- Supprimer un élément

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

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.remove('lastName');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

Immuabilité - Immer

- Problème avec Immutable.js, il est nécessaire d'exécuter du code pour désérialiser du JSON
- Une autre approche avec Immer.js qui va "traduire" du code muable en code immuable
- Cela va avoir un impact sur les performances mais le code sera beaucoup plus lisible et simple à maintenir
- Bench pour mettre à jour 5000 objets dans un tableau de 50000 :

Immuabilité - Immer Tableaux

- Ajouter à la fin

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  draft.push('Jean');
});

console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  draft.unshift('Jean');
});

console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Tableaux

- Ajouter à un indice donné

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  const index = 1;
  draft.splice(index, 0, 'Jean')
});

console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Tableaux

- Modifier un élément

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  const index = 1;
  draft[index] = 'Jean';
});

console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Tableaux

- Supprimer un élément

```
import { produce } from 'immer';

const firstNames = ['Romain', 'Edouard'];

const newfirstNames = produce(firstNames, (draft) => {
  const index = 1;
  draft.splice(index, 1);
});

console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```

Immuabilité - Immer Objets

- Ajouter un élément

```
import { produce } from 'immer';

const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

const newContact = produce(contact, (draft) => {
  draft.city = 'Paris';
});

console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

Immuabilité - Immer Objets

- Modifier un élément

```
import { produce } from 'immer';

const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

const newContact = produce(contact, (draft) => {
  draft.firstName = 'Thomas';
});

console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

Immuabilité - Immer Objets

- Supprimer un élément

```
import { produce } from 'immer';

const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

const newContact = produce(contact, (draft) => {
  delete draft.lastName
});

console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

Immuabilité - Immer Objets

- Immer est particulièrement utile lorsque qu'il y a des objets ou tableaux imbriqués
- Sans Immer :

```
const contacts = [
  { firstName: 'Romain', address: { city: 'Paris' } },
  { firstName: 'Edouard', address: { city: 'Lille' } },
  { firstName: 'Brice', address: { city: 'Nice' } },
];

const newContacts = [
  ...contacts.slice(0, 1),
  {
    ...contacts[1],
    address: {
      ...contacts[1].address,
      city: 'Bordeaux'
    }
  },
  ...contacts.slice(1 + 1),
]

console.log(JSON.stringify(newContacts));
// [
//   {"firstName":"Romain","address":{"city":"Paris"}},
//   {"firstName":"Edouard","address":{"city":"Bordeaux"}},
//   {"firstName":"Brice","address":{"city":"Nice"}}
// ]
console.log(contacts === newContacts); // false
```

Immuabilité - Immer Objets

- Immer est particulièrement utile lorsque qu'il y a des objets ou tableaux imbriqués
- Avec Immer :

```
import { produce } from 'immer';

const contacts = [
  { firstName: 'Romain', address: { city: 'Paris' } },
  { firstName: 'Edouard', address: { city: 'Lille' } },
  { firstName: 'Brice', address: { city: 'Nice' } },
];

const newContacts = produce(contacts, (draft) => {
  draft[1].address.city = 'Bordeaux';
});

console.log(JSON.stringify(newContacts));
// [
//   {"firstName":"Romain","address":{"city":"Paris"}},
//   {"firstName":"Edouard","address":{"city":"Bordeaux"}},
//   {"firstName":"Brice","address":{"city":"Nice"}}
// ]
console.log(contacts === newContacts); // false
```

formation.tech

Optimisation

Optimisation - Introduction

- Plusieurs leviers sont disponibles pour optimiser une application React
  - diminuer l'empreinte mémoire
  - diminuer le cout CPU notamment lors des render
  - diminuer le poids de l'application lors du chargement
- Identifier les composants à risque notamment les listes avec React Developper Tools ou Chrome DevTools
- Les performances des différents frameworks JavaScript : https://github.com/krausest/js-framework-benchmark

Optimisation - Keys

- Au moment de la réconciliation, React va comparer la version précédente du Virtual DOM d'un composant avec la version actuelle (juste après l'appel à render)
- Avec un tableau passé en props ou state, si une valeur est insérée au début, l'ensemble des éléments du DOM devront être mis à jour.
- Pour éviter cela, il faut passer au Virtual DOM un paramètre key dans les props lui permettant d'établir un lien entre l'élément du Virtual DOM et l'élément du DOM
- Choisir une valeur unique, et non modifiée en cas de mise à jour de l'élément (id de la database, uuid généré à la création de l'élément)

```
function DefinitionList() {
  const abbrs = { JS: "JavaScript", CSS: "Cascading Style Sheets" };
  return (
    <dl>
      {Object.entries(abbrs).map(([term, definition]) => (
        <ListItem key={term} term={term} definition={definition} />
      ))}
    </dl>
  );
}
```

Optimisation - Cleanup

- Un composant qui appelle un API en 2 partie création/arrêt doit nettoyer cet API au moment où le composant disparaît (componentWillUnmount, fonction de nettoyage retournée par un effet) :

```
class Clock extends Component<Props, State> {
  state: Readonly<State> = {
    now: new Date(),
  };

  componentDidMount(): void {
    setInterval(() => {
      this.setState({
        now: new Date(),
      });
    }, 1000);
  }

  render(): ReactNode {
    const { now } = this.state;
    return <div className="Clock">{now.toLocaleTimeString()}</div>;
  }
}
```

Optimisation - shouldComponentUpdate

- Lorsque que le state ou les props d'un éléments sont mis à jour, une cascade de render va s'effectuer pour les composants enfants
- Il est possible de bloquer les render liés à l'update d'un élément en créant une méthode shouldComponentUpdate sur l'élément

```
shouldComponentUpdate(nextProps) {
  return this.props.todos !== nextProps.todos;
}
```

Optimisation - PureComponent

- Un composant "pur" est un composant contenant une méthode shouldComponentUpdate vérifiant que chacune des propriétés est différente de la valeur précédente
- Lorsque qu'on utilise la classe PureComponent, il faudra dont mettre à jour les tableaux et les objets de façon "immuable"

```
class TodoList extends PureComponent {
  render() {
    const todoItems = this.props.todos.map((todo) => (
      <TodoItem key={todo.id} todo={todo}
                onDelete={() => this.props.onDelete(todo)} />
    ));

    return (
      <div className="TodoList">
        {todoItems}
      </div>
    );
  }
}
```

Optimisation - Mémoïsation

- La mémoïsation est une technique de programmation qui consiste à mettre en cache le retour d'une fonction lorsque ses paramètres sont inchangés : https://fr.wikipedia.org/wiki/Mémoïsation
- Lodash contient par exemple une fonction memoize

```
const nbs = (new Array(1_000_000)).fill(0).map(() => Math.random());

function findLowerCount(arrayNbs, val) {
  return arrayNbs.filter((el) => el < val).length;
}

// Sans memoisation
console.time('findLowerCount');
console.log(findLowerCount(nbs, 0.5)); // 498630
console.timeEnd('findLowerCount'); // 71.366ms

console.time('findLowerCount');
console.log(findLowerCount(nbs, 0.5)); // 498630
console.timeEnd('findLowerCount'); // 60.217ms

console.time('findLowerCount');
console.log(findLowerCount(nbs, 0.5)); // 498630
console.timeEnd('findLowerCount'); // 43.323ms
```

Optimisation - Mémoïsation

```javascript
const { memoize } = require('lodash');

function findLowerCount(arrayNbs, val) {
  return arrayNbs.filter((el) => el < val).length;
}

// Avec memoisation
const findLowerCountMemo = memoize(findLowerCount);
console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5));
console.timeEnd('findLowerCountMemo'); // 71.366ms

console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5)); // pas de rappel
console.timeEnd('findLowerCountMemo'); // 0.102ms

console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5)); // pas de rappel
console.timeEnd('findLowerCountMemo'); // 0.045ms
```

Optimisation - Limiter le nombre d'éléments

- Lighthouse recommande de ne pas dépasser 800 noeud par page : https://developer.chrome.com/docs/lighthouse/performance/dom-size/
- Pour éviter des pages trop conséquentes on peut avoir recours :
  - Wizard / Funnel pour les formulaires : https://react-hook-form.com/advanced-usage#WizardFormFunnel
  - à la pagination ou Virtual Scroll pour les listes : https://react-window.vercel.app/

formation.tech

Internationalisation (i18n)

i18n - Principales Bibliothèques

- Les 2 principales bibliothèques d'internationalisation sous React sont
  - react-intl
  - react-i18next
- Nous allons voir react-i18next qui propose certaines fonctionnalités plus avancées comme l'extraction des clés de traduction

i18n - Installation

- react-i18next dépend de i18next
  `npm i i18next react-i18next`
- De nombreux plugins existent
  https://www.i18next.com/overview/plugins-and-utils
- i18next-xhr-backend
  Permet de récupérer les clés de traduction avec des requêtes XHR
  `npm i i18next-xhr-backend`

i18n - Configuration

- Dans le fichier index/main

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import i18nextXhrBackend from 'i18next-xhr-backend';

i18n
  .use(initReactI18next)
  .use(i18nextXhrBackend)
  .init({
    lng: 'fr',
    fallbackLng: 'fr',
    interpolation: {
      escapeValue: false, // react echappe déjà
    },
  });
```

i18n - Fichiers de traductions

- Les fichiers de traduction doivent être placés dans public/locales/{{lng}}/{{namespace}}.json

```
// public/locales/en/translation.json
{
  "title": "Welcome to react using react-i18next",
  "description": {
    "part1": "To get started, edit <1>src/App.js</1> and save to reload.",
    "part2": "Switch language between english and german using buttons above."
  }
}
```

i18n - Appel des traductions

- Pour appeler les traductions on peut utiliser un hook, un composant ou un hoc
- Exemple avec le hook :

```
import React from 'react';
import { useTranslation } from 'react-i18next';

export function TodoCount({ todos }) {
  const { t } = useTranslation();
  return (
    <div className="TodoCount">
      <h2>{t('todos.title')}</h2>
      {todos.length}
      {t('todos.remaining', { count: todos.length })} {/* pour les pluriels */}
    </div>
  );
}
```

i18n - Changer de langue

- Changer de langue

```
i18n.changeLanguage('es').then(() => {
  console.log('changed to es');
});
```

i18n - Extraction des clés

- Installation
  `npm i -D i18next-parser`
- Script NPM

```
{
  "scripts": {
    "i18n-extract": "i18next"
  }
}
```

i18n - Extraction des clés

- Configuration
  Formats possibles : js, mjs, json, ts, yaml, yml

```
// i18next-parser.config.ts (à la racine du projet)
import { UserConfig } from "i18next-parser";

const config: UserConfig = {
  locales: ['en', 'fr'],
  output: 'public/locales/$LOCALE/$NAMESPACE.json',
  input: ['src/**/*.{js,jsx,ts,tsx}', '!src/**/*.spec.{js,jsx,ts,tsx}'],
  sort: true,
  verbose: true,
};


export default config;
```

formation.tech

Styling

Styling - Introduction

- Pour mettre en forme un composant on peut utiliser différentes approches :
  - importer un CSS avec les outils de build (webpack, vite...)
  - transformer en CSS via un préprocesseur (SASS, LESS, Stylus...)
  - limiter les conflits avec les modules CSS
  - utiliser des bibliothèques de CSS in JS
  - utiliser des classes dynamiques
  - utiliser des propriétés ou des valeurs dynamiques

Styling - CSS et Préprocesseurs

- Les outils de build supportent en général l'import de fichier CSS depuis un fichier JS :

```
import './pokemon-card.css';
```

- Il est également possible de configurer un préprocesseur comme SASS, LESS ou Stylus qui ajouteront des fonctionnalités au language (fonction, nested rules, boucles...)

```
import './pokemon-card.scss';
```

Styling - CSS Modules

- En général avec les outils de build, si le fichier est suffixé par .module.css ou .module.ext, le noms de classe seront renommé pour éviter les conflits

```
import styles from './pokemon-card.module.css';
```

- Il faudra alors importer un objet styles qui fera le mapping entre le nom de classe d'origine et le nom généré

Styling - CSS in JS

- Une alternative populaire au CSS "buildé"
- Un choix de bibliothèques immense : https://github.com/MicheleBertoli/css-in-js
- Principales bibliothèques :
  - styled-components
  - emotion
  - styled-jsx
- Certains frameworks de composants d'UI imposent ces bibliothèques (MUI)

Styling - CSS in JS

- Avec Emotion

```
import { css } from '@emotion/react';

export default function MyComponent() {
  const color = 'white';
  return (
    <div
      css={css({
        padding: '32px',
        backgroundColor: 'hotpink',
        fontSize: '24px',
        borderRadius: '4px',
        '&:hover': {
          color,
        },
      })}
    >
      Hover to change color.
    </div>
  );
}
```

```
▶<head> … </head>
▼<body>
  ▼<div id="root">
    ▼<div>
      <div class="css-jn1amr-MyComponent">Hover to change color
    </div>
  </div>
  <script type="module" src="/src/main.tsx"></script>
</body>
</html>
```

html body div#root div div.css-jn1amr-MyComponent

Styles Computed Layout Event Listeners DOM Breakpoints Prope

Filter :hov .

```
element.style {
}
.css-jn1amr-MyComponent:hover {
  color: white;
}
.css-jn1amr-MyComponent {
  padding: ▶ 32px;
  background-color: hotpink;
  font-size: 24px;
  border-radius: ▶ 4px;
}
```

Styling - classnames

- Ajouter des classes de manière conditionnelle avec classnames :

```
import classNames from 'classnames';

classNames('foo', 'bar'); // => 'foo bar'
classNames('foo', { bar: true }); // => 'foo bar'
classNames({ 'foo-bar': true }); // => 'foo-bar'
classNames({ 'foo-bar': false }); // => ''
classNames({ foo: true }, { bar: true }); // => 'foo bar'
classNames({ foo: true, bar: true }); // => 'foo bar'

// lots of arguments of various types
classNames('foo', { bar: true, duck: false }, 'baz', { quux: true }); // => 'foo bar baz
quux'

// other falsy values are just ignored
classNames(null, false, 'bar', undefined, 0, 1, { baz: null }, ''); // => 'bar 1'
```

Styling - style

- Lorsque la propriété ou la valeur nécessite d'exécuter du code JS on utilisera la propriété style
- Par rapport à la propriété du DOM elle est légèrement réécrite (pas besoin de spécifier les unités en 'px' notamment) :
  https://react.dev/reference/react-dom/components/common#common-props
  https://react.dev/reference/react-dom/components/common#applying-css-styles

formation.tech

Animations/Transitions

Animations/Transitions - react-transition-group

- Bibliothèque intégrée à React à l'origine puis confié à la communauté
- Inspirée par ng-animate d'AngularJS
- Installation
  npm i react-transition-group

Animations/Transitions - react-transition-group

- Exemples

```
function App() {
  const [inProp, setInProp] = useState(false);
  return (
    <div>
      <CSSTransition in={inProp} timeout={200} classNames="my-node">
        <div>
          {"I'll receive my-node-* classes"}
        </div>
      </CSSTransition>
      <button type="button" onClick={() => setInProp(true)}>
        Click to Enter
      </button>
    </div>
  );
}
```

Animations/Transitions - Principales Bibliothèques

- Quelques resources
  - Comparatif de 20 bibliothèque d'animation pour React
    https://bashooka.com/coding/20-useful-react-animation-libraries/
  - Un même exemple écrit avec 7 bibliothèques d'animations
    https://github.com/aholachek/react-animation-comparison

formation.tech

Tests Automatisés

Tests Automatisés - Introduction

- Comment tester son code ?
  - Manuellement : une personne effectue les tests
  - Automatiquement : les tests ont été programmés
- Historique
  - à partir de 1989 en Smalltalk et le framework SUnit
  - à partir de 1997 en Java avec JUnit
  - à partir de 2004 dans le navigateur avec Selenium

Tests Automatisés - Pourquoi ?

- Pourquoi automatiser les tests ?
  - plus l'application grandit, plus le risque d'introduire une régression est grand
    ex: modifier une fonction qui est partagé par différentes
  - tester manuellement à chaque itération prendra à terme plus de temps qu'écrire le code du test
  - les tests automatisés peuvent se lancer sur différentes plate-formes et navigateurs très simplement
  - les tests aident à la compréhension du code, les lire permet de comprendre des comportements qui n'ont pas toujours été documentés
- Pourquoi continuer de tester manuellement ?
  - certains tests peuvent être simple à faire manuellement mais compliqués à automatiser (drag-n-drop...)
  - automatiser permet d'avoir accès à des choses inaccessibles manuellement (bouton caché par une popup...)

Tests Automatisés - Types de tests

- Types de tests :
  - tests de code statiques / linters
  - tests de code dynamiques / tests unitaires...
  - tests de déploiement
  - tests de sécurité (pentests)
  - tests de montée en charge
  - ...

Tests Automatisés - Pyramide des tests

- 3 types de tests automatisés au niveau code côté Front :
  - Test unitaire
    Permets de tester les briques d'une application (classes / fonctions)
  - Test d'intégration
    Teste que les briques fonctionnent correctement ensembles
  - Test End-to-End (E2E)
    Vérifie l'application dans le client
- Une pyramide
  - plus le test est haut plus il est lent
  - plus le test est haut plus il coute cher

Tests Automatisés - Quand exécuter ?

- Quand exécuter ?
  - tout le temps si on arrive à maintenir des tests performants (max 1-2 minutes)
  - avant un commit
  - avant un push
  - sur une plateforme d'intégration ou de déploiement continu (CI/CD)

Tests Automatisés - Organisation

- Ou placer ses tests ?
  - dans le même répertoire que le code testé
  - dans un répertoire *test* en préservant l'arborescence du répertoire *src*
  - dans un répertoire *test* sans lien avec l'arborescence

formation.tech

Jest / Vitest

Jest - Introduction

- Framework de test créé en 2014 par Facebook
- Sous Licence MIT depuis septembre 2017
- Permet de lancer des tests :
  - unitaires / d'intégration (dans Node.js)
  - fonctionnels / E2E (via Puppeteer ou PlayWright)
- Peut s'utiliser avec ou sans configuration
- Les tests se lancent en parallèle dans les Workers Node.js
- Intègre par défaut :
  - Calcul de coverage (via Istanbul)
  - Mocks (natifs ou en installant Sinon.JS)
  - Snapshots

Jest - Installation

- Installation

```
npm install --save-dev jest
yarn add --dev jest
```

Jest - Hello, world !

- Sans configuration, les tests doivent se trouver dans un répertoire __tests__, ou bien se nommer *.test.js ou *.spec.js

```
// src/hello.js
const hello = (name = 'World') => `Hello ${name} !`;

module.exports = hello;
```

```
// __tests__/hello.js
const hello = require('../src/hello');

test('Hello, world !', () => {
  expect(hello()).toBe('Hello World !');
  expect(hello('Romain')).toBe('Hello Romain !');
});
```

Jest - Lancements des tests

- Si Jest localement
  `node_modules/.bin/jest`
- Si Jest globalement
  `jest`
- Avec un script test dans package.json
  `npm run test`
  `npm test`
  `npm t`

```
// package.json
{
  "devDependencies": {
    "jest": "^22.0.6"
  },
  "scripts": {
    "test": "jest"
  }
}
```

```
MacBook-Pro:hello-jest romain$ node_modules/.bin/jest
 PASS  __tests__/hello.js
  ✓ Hello, world ! (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.701s, estimated 1s
Ran all test suites.
```

Jest - Watchers

- En mode Watch

```
node_modules/.bin/jest --watchAll
jest --watchAll
npm t -- --watchAll
```

```
MacBook-Pro:hello-jest romain$ npm t -- --watchAll
 PASS  __tests__/hello.js
 PASS  __tests__/calc.js

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.65s, estimated 1s
Ran all test suites.

Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Jest - Coverage

- Avec calcul du coverage
  `node_modules/.bin/jest --coverage`
  `jest --coverage`
  `npm t -- --coverage`

- Par défaut le coverage s'affiche dans la console et génère des fichiers Clover, JSON et HTML dans le dossier coverage

```
MacBook-Pro:hello-jest romain$ npm t -- --coverage
 PASS  __tests__/calc.js
 PASS  __tests__/hello.js

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.722s, estimated 1s
Ran all test suites.
----------|----------|----------|----------|----------|----------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
All files |    86.67 |      100 |       60 |      100 |                |
 calc.js  |    83.33 |      100 |       50 |      100 |                |
 hello.js |      100 |      100 |      100 |      100 |                |
----------|----------|----------|----------|----------|----------------|
```

Jest - Partager des variables entre les tests

- On peut utiliser la portée de closure

```
describe('A suite is just a function', () => {
  let a;

  test('and so is a spec', () => {
    a = true;

    expect(a).toBe(true);
  });
});
```

Jest - Hooks

- Dans une suite de tests, certaines méthodes seront appelées automatiquement durant la vie du test
  - beforeEach, avant chaque test de la suite (avant chaque *test*)
  - afterEach, après chaque test
  - beforeAll, avant le premier test de la suite (avant le premier *test*)
  - afterAll, après le dernier test de la suite

```
describe('A suite with some shared setup', () => {
  let foo = 0;
  beforeEach(() => {
    foo += 1;
  });
  afterEach(() => {
    foo = 0;
  });
  beforeAll(() => {
    foo = 1;
  });
  afterAll(() => {
    foo = 0;
  });
});
```

Jest - Désactiver les tests

- Pour désactiver certains tests on peut :
  - les commenter
  - utiliser la fonction *test.skip* au lieu de *test*
  - utiliser la fonction *describe.skip* au lieu de *describe*

```
describe('sum function', () => {
  test.skip('should add positive number', () => {
    expect(sum(1, 2)).toEqual(3);
  });
  test('should convert strings to numbers', () => {
    expect(sum('1', '2')).toEqual(3);
  });
});
```

```
➜  hello-jasmine npm t

> @ test /Users/romain/Desktop/prepa-angular-tests/hello-jasmine
> jasmine

Randomized with seed 01882
Started
.


Ran 1 of 5 specs
1 spec, 0 failures
Finished in 0.011 seconds
```

Jest - Désactiver les tests

- On peut également forcer l'exécution d'un test (et pas les autres) avec
  - *test.only* pour un test en particulier
  - *describe.only* pour un groupe de test

Jest - Tester les erreurs

- Pour tester les erreurs on utilise
  - un callback dans le expect
  - toThrow

```
export function throwError() {
  throw new Error('Error Message');
}
```

```
import { expect, test } from 'vitest';
import { throwError } from './throwError';

test('throwError function', () => {
  expect(() => throwError()).toThrow();
  expect(() => throwError()).toThrow('Error Message');
  expect(() => throwError()).toThrow(/error message/i);
  expect(() => throwError()).toThrow(Error);
  expect(() => throwError()).toThrow(new Error('Error Message'));
});
```

Jest - Mocks

- Créer une fonction de test

```
export function withCallback(cb: (val: string) => void) {
  cb('ABC');
}
```

```
import { expect, test, vitest } from 'vitest';
import { withCallback } from './withCallback';

test('withCallback function', () => {
  const mockFn = vitest.fn();

  withCallback(mockFn);

  expect(mockFn).toHaveBeenCalled();
  expect(mockFn).toHaveBeenCalledOnce();
  expect(mockFn).toHaveBeenCalledTimes(1);
  expect(mockFn).toHaveBeenCalledWith('ABC');
});
```

Jest - Mocks

- Avec une implémentation

```
export function withCallbackReturn(cb: (val: string) => string): string {
  return cb('ABC');
}
```

```
import { expect, test, vitest } from 'vitest';
import { withCallbackReturn } from './withCallbackReturn';

test('withCallbackReturn function', () => {
  const mockFn = vitest.fn().mockImplementation(() => 'XYZ');
  // en raccourci :
  // const mockFn = vitest.fn().mockReturnValue('XYZ');

  const val = withCallbackReturn(mockFn);

  expect(mockFn).toHaveBeenCalled();
  expect(mockFn).toHaveBeenCalledOnce();
  expect(mockFn).toHaveBeenCalledTimes(1);
  expect(mockFn).toHaveBeenCalledWith('ABC');
  expect(val).toBe('XYZ');
});
```

Jest - Mocks

- Pour espionner une méthode d'un objet

```
export function secondsFromNow(timestamp: number) {
  return Date.now() - timestamp;
}
```

```
import { expect, test, vitest } from 'vitest';
import { secondsFromNow } from './secondsFromNow';

test('secondsFromNow function', () => {
  vitest.spyOn(Date, 'now').mockReturnValue(new Date(2023, 9, 1, 0, 0, 0, 30).getTime())

  expect(secondsFromNow(new Date(2023, 9, 1, 0, 0, 0).getTime())).toBe(30)
});
```

Jest - Mocks

- Pour espionner un module

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

```
import { fetchUsers } from './fetchUsers';

export async function listUsers() {
  const users = await fetchUsers();
  return users;
}
```

```
import { expect, test, vitest } from 'vitest';
import { listUsers } from './listUsers';
import { fetchUsers } from './fetchUsers';

vitest.mock('./fetchUsers');

test('listUsers function', async () => {
  vitest.mocked(fetchUsers).mockResolvedValue([{id: 1, name: 'Toto'}])
  const users = await listUsers();
  expect(users).toEqual([{id: 1, name: 'Toto'}])
});
```

Jest - Tester les timers

- La fonction jest.useFakeTimers() transforme les timers (setTimeout, setInterval...) en mock

```
export function timeout(delay: number, arg: any) {
  return new Promise((resolve) => {
    setTimeout(resolve, delay, arg);
  });
}
```

```
import { expect, test, vitest } from 'vitest';
import { timeout } from './timeout';

// ✓ hello function 1002ms
test('hello function', async () => {
  const val = await timeout(1000, 'ABC');
  expect(val).toBe('ABC');
});

// ✓ hello function
test('hello function', async () => {
  vitest.useFakeTimers();
  const promise = timeout(1000, 'ABC');
  vitest.advanceTimersByTime(1000);
  const val = await promise;
  expect(val).toBe('ABC');
  vitest.useRealTimers();
});
```

formation.tech

Tests React

Tests React - Introduction

- Le plus simple pour exécuter ses tests avec React : vitest
- Vitest utilise le même API que Jest et supporte JSX et TypeScript par défaut

Tests React - Hello

- Pour tester ses composants Jest/Vitest vont lancer les tests dans Node.js (via des Workers)
- Il faut donc émuler les APIs Web via des bibliothèques comme JSDOM ou plus moderne Happy DOM
- vitest --dom

```
// npx vitest --dom
test('Hello renders (ReactDOM)', () => {
  const rootEl = document.createElement('div');
  createRoot(rootEl).render(<Hello />);
});
```

Tests React - 3 façons de tester

- Plusieurs options pour tester unitairement :
  - Utiliser l'object document fourni par JSDOM / Happy DOM
  - Utiliser react-test-renderer qui va faire le rendu des composants sous forme d'objet literal
  - Utiliser des bibliothèques haut niveau comme Enzyme ou Testing Library (recommandé)

```
// npx vitest --dom
test('Hello renders (ReactDOM)', () => {
  const rootEl = document.createElement('div');
  createRoot(rootEl).render(<Hello />);
});

test('Hello renders (react-test-renderer)', () => {
  const renderer = ReactTestRenderer.create(
    <Hello />
  );
});

// npx vitest --dom
test('Hello renders (testing-library)', () => {
  render(<Hello />);
});
```

Tests React - Base

- Pour tester un composant React il faut en faire le rendu

```
// src/App.test.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

- 2 inconvénients ici :
  - Nécessite que document existe (exécuter les tests dans un navigateur ou utiliser des implémentations de document côté Node.js comme JSDOM)
  - Les composants enfants du composant testé seront également rendu, le test n'est pas un test unitaire mais un test d'intégration

Tests React - Snapshot Testing

- Facebook fourni un paquet npm pour simplifier les tests : react-test-renderer
- Ici on fait un simple Snapshot, c'est à dire une capture du rendu du composant, si lors d'un test futur le rendu est modifié le test échoue

```
import React from 'react';
import renderer from 'react-test-renderer';
import { Hello } from './Hello';

test('it renders like last time', () => {
  const tree = renderer
    .create(<Hello />)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

Tests React - Shallow Rendering

- On peut également faire appel à ShallowRenderer qui ne va faire qu'un seul niveau de rendu, et donc rendre le test unitaire

```
// src/App.test.js
import React from 'react';
import App from './App';
import ShallowRenderer from 'react-test-renderer/shallow';
import { Hello } from './Hello';
import { CounterButton } from './CounterButton';

it('renders without crashing', () => {
  const renderer = new ShallowRenderer();
  renderer.render(<App />);
  const result = renderer.getRenderOutput();

  expect(result.type).toBe('div');
  expect(result.props.children).toEqual([
    <Hello firstName="Romain" />,
    <hr />,
    <CounterButton/>,
  ]);
});
```

Tests React - Enzyme

- Facebook recommande également l'utilisation de la bibliothèque Enzyme, créée par AirBnB.
- Elle fourni un API haut niveau (proche de jQuery) pour manipuler les tests des composant

```
import React from 'react';
import { Hello } from './Hello';
import { shallow } from 'enzyme';

test('it renders without crashing with enzyme', () => {
  shallow(<Hello />);
});

test('it renders without crashing with enzyme', () => {
  const wrapper = shallow(<Hello />);
  expect(wrapper.contains(<div>Hello  !</div>)).toEqual(true);
});

test('it renders without crashing with enzyme', () => {
  const wrapper = shallow(<Hello firstName="Romain"/>);
  expect(wrapper.contains(<div>Hello Romain !</div>)).toEqual(true);
});
```

Tests React - Tester des événements

```
import React, { Component } from 'react';

export class CounterButton extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }

  render() {
    return (
      <button onClick={this.handleClick}>{this.state.count}</button>
    );
  }
}
```

Tests React - Tester des événements

- Avec Enzyme :

```
import React from 'react';
import { CounterButton } from './CounterButton';
import { shallow } from 'enzyme';

test('it renders without crashing', () => {
  shallow(<CounterButton />);
});

test('it contains 0 at first rendering', () => {
  const wrapper = shallow(<CounterButton />);
  expect(wrapper.text()).toBe('0');
});

test('it contains 1 after click', () => {
  const wrapper = shallow(<CounterButton />);
  wrapper.simulate('click');
  expect(wrapper.text()).toBe('1');
});
```

Tests React - Tester des événements

- Avec Testing Library :

```
import { render, screen, fireEvent } from "@testing-library/react";
import { CounterButton } from "./Counter";
import { expect, test } from "vitest";

// npx vitest --dom
test('CounterButton (testing-library)', () => {
  render(<CounterButton />);
  const button = screen.getByRole('button');

  expect(button.innerText).toContain('0');

  fireEvent.click(button);

  expect(button.innerText).toContain('1');
});
```