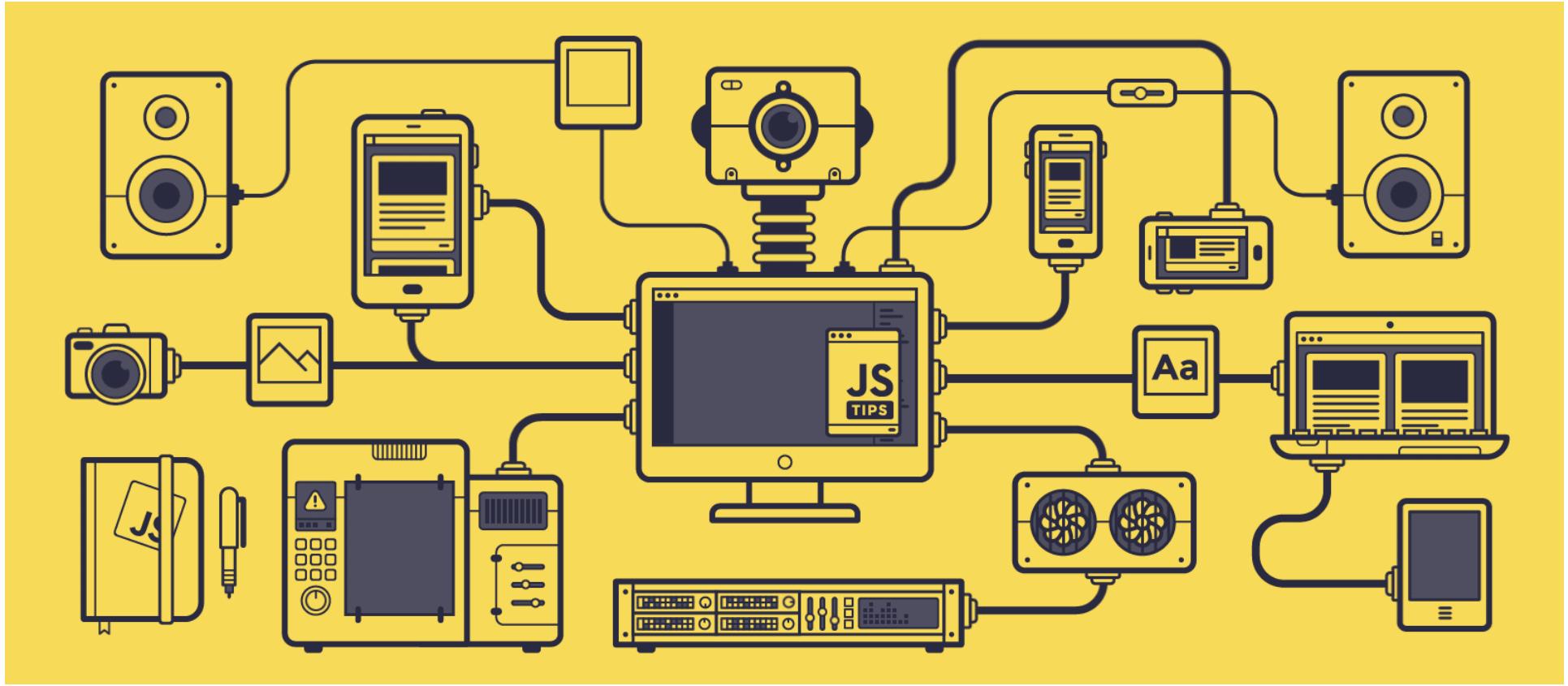
FORMATION REACT 19

Juin 2025

Mustapha Bargach

PRÉSENTATION



1. Présentation des outils
2. Javascript et Typescript
3. Écosystème
4. React
5. Architecture logicielle

POURQUOI REACT ?

- Pour l'utilisateur
  - Responsive Web Design
  - Performance
  - RIA

- Pour le développeur
  - Réduction des coûts de développement
  - Responsive Web Design
  - Lisibilité : Sémantique et simplicité
  - Modularité
  - Portabilité
  - Documentation et support de la communauté
  - Testabilité

ET POURQUOI PAS LES AUTRES ?

OUTILS

NODE ET NPM

Plateforme JavaScript

Intègre un serveur HTTP

Implémentation de CommonJS

Possibilité d’applications full stack JS

Gestionnaire des dépendances et de la configuration du projet

Suit un formalisme : package.json

Méta-données du projet

Dépendances de développement et de production

Gestion des scripts

OUTILS

GIT

Logiciel de gestion de version décentralisée

Forte capacité de débrancher et de fusionner

Fonctionnalités dédiées au développement communautaire

Performance

OUTILS

LE REPO

https://github.com/MoosArga/formation_react

OUTILS

IDE

Produit JetBrains

IDE à part entière, supporte des fonctionnalités avancées de développement

Payant

Produit Microsoft

Entre l'éditeur et l'IDE

Gratuit

Customisable par plugins

1. Présentation des outils
2. **Javascript et Typescript**
3. Écosystème
4. React
5. Architecture logicielle

JAVASCRIPT / TYPESCRIPT

UN PEU D'HISTOIRE

JAVASCRIPT / TYPESCRIPT

ECMASCRIPT

- Standard de spécifications mise en œuvre notamment pour Javascript
- Première version en juin 1997
- Dernière version ES 15 (2024), prochaine version ES Next en cours de finalisation
- Transcompilateur, tel que Babel.js, pour définir une cible de développement et une cible de production

JAVASCRIPT / TYPESCRIPT

PROTOTYPAGE

- Jusqu'à la version ES 5, sensiblement différent de la programmation objet
- Les prototypes sont muables à l'inverse des objets
- Les déclarations de fonction sont des prototypes
- Possibilité d'hériter d'un prototype au travers de chaines de prototypages et d'ouvrir au polymorphisme

JAVASCRIPT / TYPESCRIPT

TYPAGE

- Javascript est un langage non typé avec inférence de type
- Typescript est un sucre syntaxique permettant d'apporter une programmation orientée objet, typée fortement et proposant des types génériques
- Typescript appartient à la vision du développeur et nécessite d'être transpilé en Javascript pour le runtime
- Sourcemaps compatibles pour débugger en TypeScript

JAVASCRIPT / TYPESCRIPT

DÉCLARATIONS DE VARIABLES

- var : ancienne définition d'une variable en Javascript. Est soumis au hoisting et à l'inférence de type.

- let : définition d'une variable en Typescript et en Javascript (ES 7). N'est pas soumis au hoisting.

- const : déclaration d'une constante immuable. Respecte les mêmes paradigmes que le final d'une variable en Java

JAVASCRIPT / TYPESCRIPT

COLLECTIONS

- Array : le prototype propose un ensemble de fonctions orientés streams, comme par exemple map, reduce, filter ou encore sort.

- Map : collection indexée par clé / valeur

- Set : collection indexée par valeur unique

JAVASCRIPT / TYPESCRIPT

FALSY ET TRUTHY VALUES

- Toujours faux :
  - false
  - 0 (zero numérique)
  - '' ou "" (chaîne de caractères vide)
  - null
  - undefined
  - NaN

- Toujours vrai :
  - '0' (en tant que chaîne de caractères)
  - 'false' (en tant que chaîne de caractères)
  - [] (un tableau vide)
  - {} (un objet vide)
  - function(){} (une fonction “vide”)

JAVASCRIPT / TYPESCRIPT – TP

MANIPULATION DU DOM

- Aller sur GitHub : https://github.com/MoosArga/formation_react et cliquer sur le lien du TP
- Dans la balise div#hobbies, afficher une liste non ordonnée du tableau hobs
- Dans la fonction changeCivilite, modifier la balise div#presentation pour afficher alternativement Bonjour Madame et Bonjour Monsieur
- Modifier la fonction précédente pour changer la couleur de fond de la balise div#presentation en fonction de Bonjour Madame et Bonjour
- Supprimer l'attribut onClick du bouton et associer l'évènement du click sur le bouton directement dans le code JavaScript

JAVASCRIPT / TYPESCRIPT

SYNTAXE TYPESCRIPT

```typescript
// Déclaration d'une variable
let message: string = 'Bonjour';
const pattern: string = 'Bonjour {0}';

// Déclaration d'une fonction (hors classe)
function concatener(a: string, b: string): string {
    return a + b;
}

// Déclaration d'une classe
class Employe {

    nom: string;
    private readonly matricule: string;

    constructor(nom: string, matrictule: string) {
        this.nom = nom;
        this.matricule = matrictule;
    }

    hasMatriculeSet() {
        return !!this.matricule;
    }

}
```

JAVASCRIPT / TYPESCRIPT

SYNTAXE TYPESCRIPT

```typescript
// Déclaration d'une interface
interface Person {
    nom: string;
    prenom?: string;
}

// Absence de retour
function execute(cmd: string, option?: string): void { /*...*/ }

// Typages abstrait et implicite
let ordre: any = { };
let commandeNumber = 3;

// Typage union
function firstValue(a: string|number, b: string|number): string|number {
    return a || b;
}

// Opérateurs de non null assertion et d'optionnal chaining
let state = context!.state;
let isActive = context?.state === 1;
```

JAVASCRIPT / TYPESCRIPT

NICE HORROR SHOW

```
// Arrow functions
const f = (a,b) => a + b;
const g = (a,b) => { a + b };
const k = (a) => (b) => (a + b);

[1, 2, 3].reduce((a,b) => a + b, 0);
['a', 'b'].map((m, i) => m + i);

// Fallback et Coaslescence nulle
let labelF = cmdAddress || 'Non défini';
let labelC = cmdAddress ?? 'Non défini';

// Constructeur param property
class Animal {

    constructor(private nom: string) {}

}
```

1. Présentation des outils
2. Javascript et Typescript
3. **Écosystème**
4. React
5. Architecture logicielle

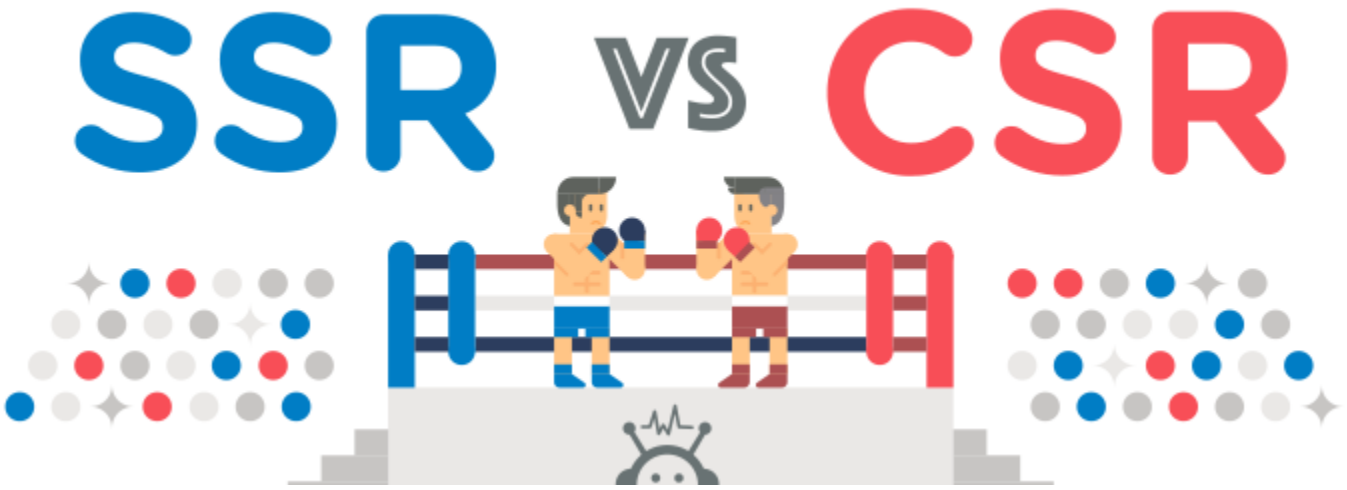
ÉCOSYSTÈME

REACT C'EST QUOI ?

- Bibliothèque **JavaScript**, créée par Facebook en 2013 (utilisé en interne dès 2011), pour créer des interfaces utilisateurs déclaratives et réactives
- Évolutions majeures :
  - 2015 : React Native
  - 2017 : nouveau moteur Fiber
  - 2019 : **Hooks** API
  - 2023+ : Server Components, Concurrent Mode
- Écosystème riche
- Futur de React :
  - Intégration poussée avec Next.js et les Server Component
  - Amélioration de la performance notamment avec *React Forget*

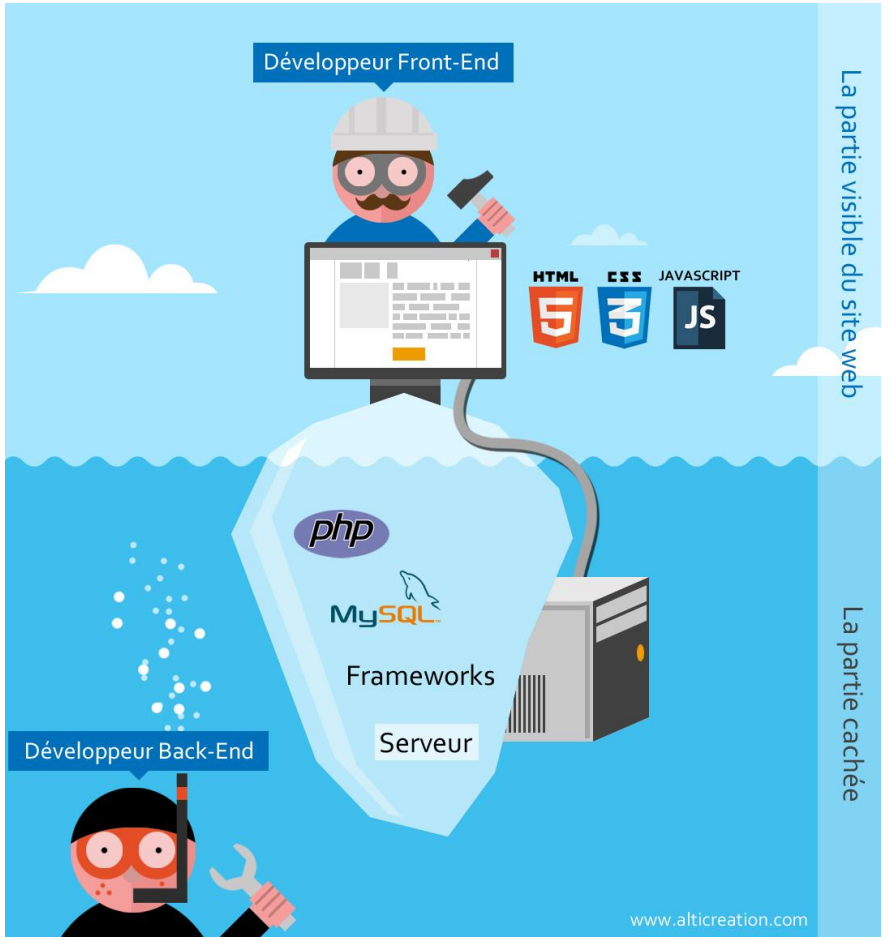
ÉCOSYSTÈME

LE RENDERING



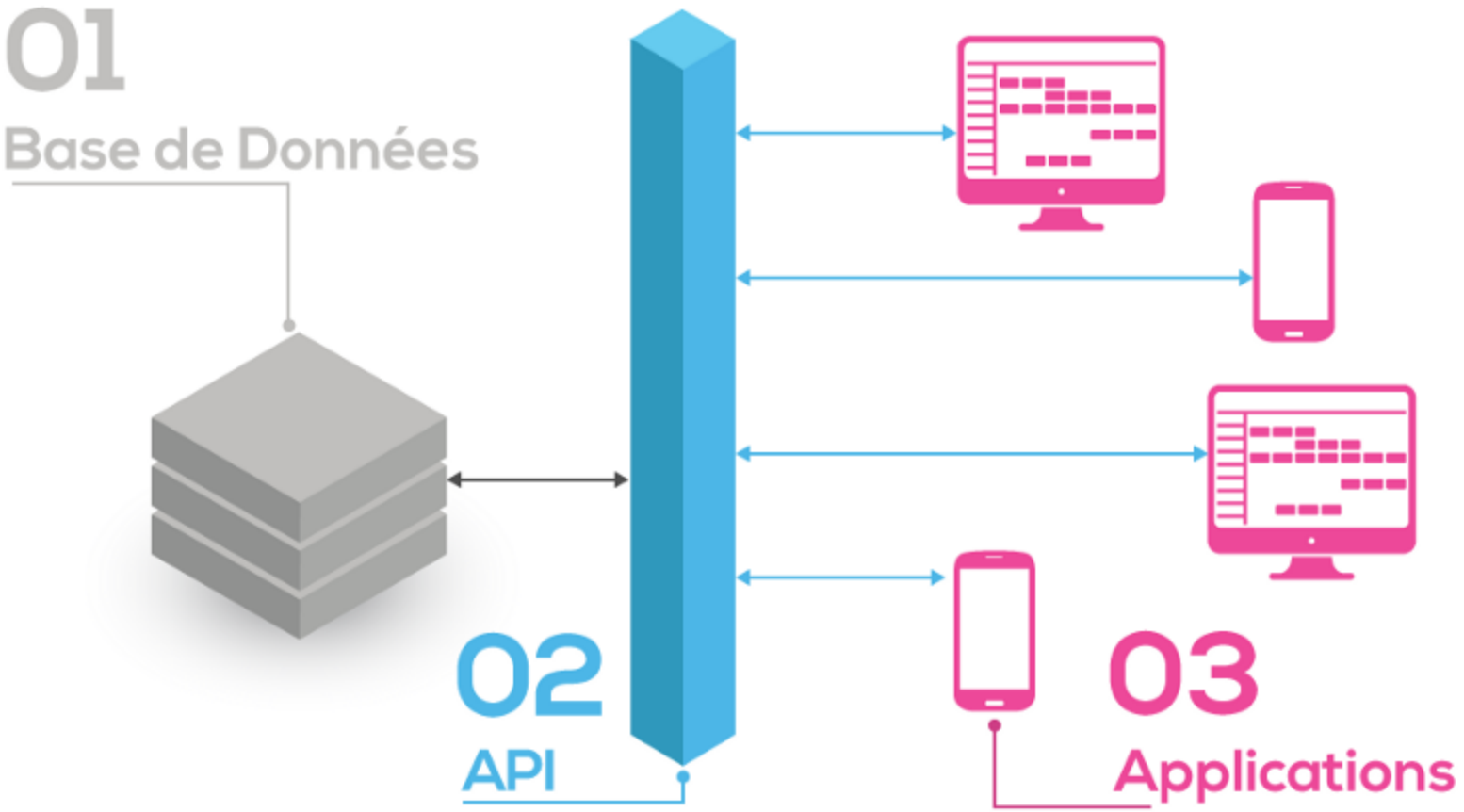
ÉCOSYSTÈME

LE RENDERING



ÉCOSYSTÈME

LES APIS



ÉCOSYSTÈME

L'UNIVERS REACT

VITE

NEXT.JS

React Native

Gatsby

React Router

AXIOS

Redux

Mobx

ÉCOSYSTÈME

VITE

- Initier une nouvelle application Vite :
  - Dans un terminal, depuis votre dossier home : *npm create vite@latest tp -- --template react-ts*
- Ouvrir un vs code dans le dossier tp et depuis un terminal exécuter :
  - *npm install*
  - *npm run dev*
- Vérifier que l'application se lance correctement sur le port 5173
- Supprimer l'entièreté du retour dans le fichier App.tsx, ne laisser que *(<></>)*
- L'application doit prendre les modifications à chaud
- Épurer l'application de son CSS et changer le titre et la langue de l'index.html

ÉCOSYSTÈME

JSON-SERVER

- Installer la librairie json-server : *npm install –g json-server*
- Copier le fichier formations.json depuis https://github.com/MoosArga/formation_react à la racine de votre projet vite
- Au sein du repository, lancer la commande : *json-server --watch formations.json*
- Depuis un navigateur, essayer les urls suivantes :
  - *http://localhost:3000/formations*
  - *http://localhost:3000/formations?nom_like=scr*

1. Présentation des outils
2. Javascript et Typescript
3. Écosystème
4. **React**
5. Architecture logicielle

1. Préambule : JSX
2. Les Composants
3. Le state management et les Hooks
4. Le Rendering et les API
5. Le router
6. Form Validation

REACT

PRÉAMBULE : JSX

- Syntaxe qui combine **JavaScript** et **balises HTML**
- Doit retourner **un seul élément parent**
- Permet d'insérer **des expressions Javascript** dans une structure HTML
- Possède des **Attributs spécifiques (className, htmlFor)** et respecte une convention en **camelCase**

```
// Interpolation
const nom = 'Formateur'
result = (
    <div>{ nom }</div>
)

// Fragment
result = (
    <>
        <div>{ nom }</div>
        <div>{ nom }</div>
    </>
)
result = (
    <Fragment>
        <div>{ nom }</div>
        <div>{ nom }</div>
    </Fragment>
)
```

```
// Binding d'attribut et d'évnènement
<input type="text" value={name} onChange={(e) => setName(e.target.value)}/>
```

```
// Attributs spécifiques
result = <label className="label-field" htmlFor="name-field">Nom</label>

// Javascript en profondeur
const menuList = ['Acceuil', 'Mon compte', 'Infos']
result = (
    <ul>
        { menuList.map((ml, i) => <li key={i}>{ ml }</li>) }
    </ul>
)
```

REACT

LES COMPOSANTS

- **Éléments de base** d'une application React
- Permettent de **découper l'interface** en blocs réutilisables
- Peuvent être soit **une fonction** (dits fonctionnels) ou **une classe** (héritage historique, quasiment plus aucun cas d'usage)
- Acceptent des **propriétés en entrée (notamment children)** en lecture seule
- Retournent **un JSX**
- Sont des **pures fonctions**
- Et respectent une convention de nommage en **PascalCase**

```
export function EmployeDogTag({ employe }) {
    return (
        <div>
            <span>Nom: { employe.nom }</span>
            <span>Matricule: { employe.matricule }</span>
        </div>
    )
}
```

REACT

LES COMPOSANTS

- Peuvent gérer **des structures conditionnelles et itératives**

```
export function EmployeTasks({ isLoading, list }) {

    // Retour conditionnel de sortie
    if (isLoading) return null;

    return (
       <ul>
        { list.map(task => <li key={task}>{ task }</li>)}
       </ul>
    )
}
```

```
export function EmployeTask({ isLoading, list }) {

    // Retour par opérateur logique ET
    return !isLoading && (
       <ul>
        { list.map(task => <li key={task}>{ task }</li>)}
       </ul>
    )
}
```

```
export function EmployeTask({ isLoading, list }) {

    // Retour par opérateur ternaire
    return !isLoading ? (
       <ul>
        { list.map(task => <li key={task}>{ task }</li>)}
       </ul>
    ) : null
}
```

```
export function EmployeTask({ isLoading, list }) {

    function Task({ taskName }) {
        return <li>{ taskName }</li>
    }

    return !isLoading ? (
       <ul>
        { list.map(task => <Task key={task} taskName={task}></Task>)}
       </ul>
    ) : null
}
```

REACT

LES COMPOSANTS : LA GESTION DU STYLE

- Style inline

```
<>
        <span style={{width: 200, fontSize: '18px'}}>{ employe.name}</span>
    </>
```

- Classe CSS statique

```
<>
        <span className="employe-name">{ employe.name}</span>
    </>
```

- Import de module CSS

```
import styles from './employe.module.scss';

export function Employe({ nom }) {
    return (
        <>
            <span className={styles.employeName}>{ employe.name}</span>
        </>
    )
}
```

REACT

LES COMPOSANTS : TP

- Créer un composant FormationManager dans un fichier dédié (.tsx)
  - Ce composant doit retourner un jsx comprenant 3 éléments au même niveau :
    - Un h1 avec comme texte : Mes formations
    - Une div vide
    - Une div avec la date à l'instant T (on utilisera new Date())
  - Intégrer ce composant dans le composant App de plus haut niveau
- Créer un dossier models et y créer un fichier typescript exportant une interface Formation :
  - id: string
  - nom: string
  - chargeH: number
  - typeF: string
  - note: number
- Dans le composant FormationManager, créer une liste statique de 3 formations. Itérer sur cette liste pour construire les lignes d'un tableau, à positionner dans la div vide.

REACT

LES COMPOSANTS : TP

- Créer un module css : FormationManager.module.css à côté du composant
- Pour chacune des lignes du tableau,
  - Si la formation à une charge en heures supérieur à 20, coloriser la ligne en rouge avec la classe adéquate
  - Si la formation à une charge en heures égale à 0, ne pas afficher la ligne.
- Repositionner la variable de la liste statique dans le composant App et la transmettre comme propriété du composant FormationManager

REACT

LE STATE MANAGEMENT

- **URL :** state porté par la navigation
- **Web storage :** state porté par la session, le navigateur
- **Local state :** state porté par un composant
- **Lifted state :** state porté par un composant parent (smart) qui le redistribue à ses composants enfants (dumb)
- **Derived state :** state qui est dérivé d'un autre state
- **Refs :** Référence du DOM (uncontrolled form)
- **Context :** state globale ou sur un périmètre transverse
- **Third party library :** state délégué à une libraire tierce, comme Redux, MobX

REACT

LES HOOKS

- **Sont des fonctions** qui apportent à un composant **fonctionnel** ce que les classes permettaient auparavant : **état local**, **effets**, **contexte**, …
- **Sont** introduits avec **React 16.8** pour simplifier et moderniser la gestion du code
- Utilisables uniquement dans les **composants fonctionnels**
- Permettent de **réutiliser de la logique** entre composants via des **hooks personnalisés**
- Doivent être appelés **en haut de la fonction** (pas dans des conditions ou des boucles)
- Livrés **par défaut avec React**

https://react.dev/reference/react/hooks

REACT

LES HOOKS : USESTATE

- Permet de **déclarer une variable d'état** dans un composant fonctionnel
- Retourne un **couple [valeur, setValeur]**
- Déclenche un **nouveau rendu** du composant quand la valeur change
- Peut être initialisé avec une **valeur primitive, un objet, un tableau, etc.**
- **Respecte un pattern immuable**

```
export function NameTextField() {

    const [name, setName] = useState<string>('')

    return (
        <input type="text" value={name} onChange={(e) => setName(e.target.value)}/>
    )
}
```

REACT

LES HOOKS : TP

- Créer un nouveau composant dans un fichier dédié : FormationCounter et l'afficher en fin du FormationManager
- Ce composant doit comporter les éléments suivants :
  - Un label « Compteur » à associer à l'input suivant,
  - Un input dont la valeur est associée à un state et dont le change met à jour ce même state
  - Un bouton « +1 » qui permet d'incrémenter une fois ce state : encapsuler l'incrémentation dans une fonction
- Ajouter un bouton « +3 » qui appelle 3 fois la fonction d'incrémentation, que constatez vous ?
- Comment corriger cela ?

REACT

LES HOOKS : USEEFFECT

- Permet d'exécuter du **code après le rendu** du composant
- Sert à gérer les **effets de bord** : requêtes réseau, timers, écouteurs, etc.
- S'écrit sous la forme : **useEffect(() => { … }, [dépendances])**
- Le tableau de dépendances contrôle **quand l'effet s'exécute**
  - **[] :** une seule fois après le 1er rendu
  - **[x,y] :** à chaque changement de x ou y
  - **Pas de tableau :** à chaque rendu
- Peut retourner une fonction de **cleanup** (nettoyage)

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

export function MinuteClock() {

    const [time, setTime] = useState<string>(() => (new Date()).toISOString());

    useEffect(() => {
        const intervalId = setInterval(() => setTime((new Date()).toISOString()), 60 * 1000)
        return () => {
            clearInterval(intervalId);
        }
    }, [])

    return (
        <span>{ time }</span>
    )

}
```

REACT

LES HOOKS : TP

- Dans le composant FormationCounter, modifier l'initialisation du useState
  pour récupérer la valeur du localStorage
- Toujours dans ce composant, à chaque changement du compteur, enregistrer la valeur dans le localstorage
- Que se passe t'il si le code FormationCounter est copié dans FormationManager ?
  Que constatez vous ?

REACT

LES HOOKS : USEREF

- Permet de **référencer un élément DOM** ou de **conserver une valeur persistante** entre les rendus
- Contrairement à useState, **modifier** le .current **ne déclenche pas de nouveau rendu**
- Fréquent pour accéder à **des éléments HTML** ou conserver **une valeur précédente, le mounted**, …

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

export function FocusableTextField() {

  const inputRef = useRef<HTMLInputElement | null>(null);

  const handleClick = () => {
    inputRef.current?.focus();
  };

  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus</button>
    </>
  );
}
```

REACT

LES HOOKS : TP

- Dans le composant FormationCounter, créer une valeur useRef qui permettra d'enregistrer la valeur précédente, à initialiser à null
- Ajouter un bouton de rollback à n'activer que si le compteur a été modifié une fois (par l'input ou par les incrémentations) qui permet de revenir à l'ancienne valeur (ne gérer qu'un seul retour)

REACT

LES HOOKS : CUSTOM HOOK

- Permettent de **réutiliser la logique métier** entre composants ou de **factoriser du code récurrent (useFetch, useAsync, …)**
- Sont des **fonctions** qui commencent par **use**
- Peuvent utiliser d'autres **hooks**
- Améliorent la **lisibilité**, la **modularité** et les **tests**
- **Ne retournent pas de JSX**, uniquement des valeurs/logiques

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

export interface TimeHook {
    time: number
}

export function useTime(defaultInterval: number = 60 * 1000): TimeHook {

    const [time, setTime] = useState<string>(() => (new Date()).toISOString());

    useEffect(() => {
        const intervalId = setInterval(() => setTime((new Date()).toISOString()), defaultInterval)
        return () => {
            clearInterval(intervalId);
        }
    }, [])

    return { time }
}
```

REACT

LES HOOKS : TP

- Créer un dossier hooks dans src et dedans un fichier useCompteur
- Transposer toute la logique du compteur dans un hook qui doit exposer les propriétés suivantes :
  - La valeur du compteur,
  - Une fonction de modification de la valeur,
  - Une propriété modified,
  - Une fonction de rollback

REACT

LES HOOKS : USEREDUCER

- Alternative à **useState** pour une gestion d'état **complexe** ou avec plusieurs sous-valeurs
- Permet de centraliser la logique de mise à jour dans un **réducteur (reducer)**
- Fonction **reducer** prend (**state**, **action**) et retourne un **nouveau state**
- Utile pour gérer des **états imbriqués** ou des **logiques complexes**
- Facilite la gestion d'**actions explicites** (type + payload)
- Prend aussi un **state initial** en argument

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

type State = { count: number };
type Action = { type: 'increment' } | { type: 'decrement' };

function reducer(state: State, action: Action): State {
  switch(action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

export function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </>
  );
}
```

REACT

LES HOOKS : TP

- Au sein du hook useCompteur, transformer le useState et les fonctions de retour avec un useReducer qui prendre une action et des paramètres

REACT

LES HOOKS : USEMEMO

- Mémorise le résultat d'une **fonction coûteuse**
- S'écrit **useMemo(() => { … return valeur }, [dépendances])**
- Évite les **recalculs inutiles** entre les rendus si les **dépendances ne changent pas**
- Optimise les **performances** dans certains cas précis
- À utiliser pour des **calculs lourds**, du **tri**, ou des **filtres complexes**

```
export function List({ props }) {

    const [myList, setMyList] = useState<string[]>([]);

    const listLength = myList.length;

    return (
        <span>{ listLength }</span>
    )
}
```

```
export function List({ props }) {

    const [myList, setMyList] = useState<string[]>([]);

    const listLength = useMemo<number>(() => myList.length, [myList]);

    return (
        <span>{ listLength }</span>
    )
}
```

REACT

LES HOOKS : USECALLBACK

- Mémorise une **fonction** entre les rendus
- S’écrit **useCallback((params?) => {…}, [dépendances])**
- Ne recrée la fonction que si ses **dépendances changent**
- Utile pour éviter des **re-renders inutiles** sur les enfants
- À utiliser avec des **composants mémoïsés (notamment React.memo)** ou des **effets**
- Permet de stabiliser la **référence d’une fonction**

REACT

LES HOOKS : USECALLBACK

```tsx
import { useCallback, useState } from "react";
import ProductFilter from "./ProductFilter";

type Product = { id: number; name: string; price: number };

const ProductList = () => {
  const [maxPrice, setMaxPrice] = useState(50);
  const [products] = useState<Product[]>([
    { id: 1, name: "T-shirt", price: 20 },
    { id: 2, name: "Pantalon", price: 60 },
  ]);

  const handleFilter = useCallback(() => {
    return products.filter((p) => p.price <= maxPrice);
  }, [products, maxPrice]);

  return (
    <div>
      <input
        type="range"
        value={maxPrice}
        min={0}
        max={100}
        onChange={(e) => setMaxPrice(Number(e.target.value))}
      />
      <ProductFilter onFilter={handleFilter} />
    </div>
  );
};

export default ProductList;
```

```tsx
import React from "react";

type Props = {
  onFilter: () => { id: number; name: string; price: number }[];
};

const ProductFilter = React.memo(({ onFilter }: Props) => {
  const products = onFilter();

  return (
    <ul>
      {products.map((p) => (
        <li key={p.id}>{p.name}</li>
      ))}
    </ul>
  );
});

export default ProductFilter;
```

REACT

LES HOOKS : USECONTEXT

- **Partage de données** globales entre composants sans « prop drilling »
- **Consomme** un contexte créé avec **createContext**
- Utilisé pour des **thèmes, utilisateurs, locales**, etc.
- Ne déclenche un **re-render que si la valeur du contexte change**
- Doit être utilisé **dans un composant englobé** par le provider du contexte
- Permet de **récupérer le contexte avec useContext**

REACT

LES HOOKS : USECONTEXT

```tsx
import React, { createContext, useState, useContext, ReactNode } from 'react'

type ThemeContextType = {
  theme: string
  setTheme: (value: string) => void
}

// Définition d'un contexte avec createContext (c'est le "store de données" que je veux gérer)
const ThemeContext = createContext<ThemeContextType | undefined>(undefined)

// Définition d'un composant parent permettant de définir le périmètre du contexte
export function ThemeProvider({ children }: { children: ReactNode }) {
  const [theme, setTheme] = useState('light')

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  )
}

// Récupération du contexte
export function useTheme() {
  const context = useContext(ThemeContext)
  if (!context) throw new Error('useTheme must be used within ThemeProvider')
  return context
}
```

```tsx
<>
  <ThemeProvider>
    <ThemeSwitcher />

      ...
      ...
  </ThemeProvider>
</>
```

```tsx
import { useTheme } from './ThemeContext'

function ThemeSwitcher() {
  const { theme, setTheme } = useTheme()

  return (
    <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>
      Thème actuel : {theme}
    </button>
  )
}
```

REACT

LES HOOKS : LES OUTSIDERS

- **useTransition** : Permet de différer des mises à jour d'interface moins prioritaires pour améliorer la réactivité et donner du feedback
- **useOptimistic** : Gère un état optimiste qui prédit les résultats avant la confirmation serveur
- **useId** : Génère des identifiants stables uniques pour l'accessibilité et lier des éléments
- **useDeferredValue** : Retarde la mise à jour d'une valeur pour éviter des rendus lourds trop fréquents
- **use :** expérimental pour React Server Component, permet de récupérer la valeur d'une promesse à l'image d'un await intégré dans le cycle de rendu

REACT

LES HOOKS : LES OUTSIDERS

```tsx
import React, { useState, useTransition } from 'react'

const items = Array.from({ length: 10000 }, (_, i) => `Item ${i + 1}`)

export default function FilterList() {
  const [inputValue, setInputValue] = useState('')
  const [filteredList, setFilteredList] = useState<string[]>(items)
  const [isPending, startTransition] = useTransition()

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const value = e.target.value
    setInputValue(value) // Feedback immédiat (affichage de l'input)

    startTransition(() => {
      // Calcul lourd différé
      const filtered = items.filter((item) => item.includes(value))
      setFilteredList(filtered)
    })
  }

  return (
    <div>
      <input
        type="text"
        value={inputValue}
        onChange={handleChange}
        placeholder="Filtrer les items"
      />
      {isPending && <p>Chargement...</p>}
      <ul>
        {filteredList.map((item) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    </div>
  )
}
```

REACT

RENDERING

Phase	Description
Initial Render	Le composant est monté pour la première fois. React exécute le composant et génère le JSX.
Commit Phase	React applique les changements au DOM, crée les refs, exécute les effets useEffect avec tableau de dépendances vide.
Re-render	Lorsqu'un état (useState) ou une prop change, React ré-exécute la fonction composant pour recalculer le JSX.
Layout Effects	Exécution des effets synchrones dans useLayoutEffect avant le rendu visuel.
Passive Effects	Exécution des effets asynchrones dans useEffect après que les changements ont été commit au DOM.
Unmounting	Lorsqu'un composant est retiré, React exécute les fonctions de nettoyage retournées par les effets.

REACT

RENDERING : MAIS QUAND REACT RE-RENDER ?

- Changement de **state local, de context ou de router**
  - Les states ou les props de navigation
  - useState, useReduce et les custom hooks
  - useContext
- Modification des **props** reçues par le composant
- Parent qui **re-render** et ne mémorise pas son enfant avec **React.memo**
- **Force update** explicite

REACT

APPEL D'API

- La consommation des API se avec **fetch**, **axios** ou d'autres bibliothèques HTTP
- Les appels se font généralement dans un **useEffect** pour gérer le cycle de vie
- Les résultats **sont stockés dans un state**
- Il faut gérer l'état de chargement et d'erreur pour l'UI
- Une requête peut être annulée (avec AbortController par exemple) pour éviter des effets indésirables lors du démontage du composant
- Il est important de nettoyer les effets dans **le retour de useEffect** pour éviter les fuites mémoire
- Utiliser **async/await ou promises**
- Adapter le rendu **en fonction des états** (loading, error, data)

REACT

APPEL D'API

```tsx
import React, { useEffect, useState } from 'react'

function useFetch<T>(url: string) {
  const [data, setData] = useState<T | null>(null)
  const [loading, setLoading] = useState<boolean>(false)
  const [error, setError] = useState<string | null>(null)

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true)
      setError(null)
      try {
        const response = await fetch(url)
        if (!response.ok) throw new Error('Erreur réseau')
        const json: T = await response.json()
        setData(json)
      } catch (err: any) {
        setError(err.message)
      } finally {
        setLoading(false)
      }
    }

    fetchData()
  }, [url])

  return { data, loading, error }
}
```

REACT

TP

- Lancer le backend json-server qui a été préalablement initialisé
- Créer un nouveau hook, useFormations qui aura pour objectif de nous retourner la liste des formations et qui attend une url en entrée. Il devra récupérer la liste des forcément depuis le backend
- Afficher cette liste de formations dans le tableau en lieu et place de la liste statique actuelle. Le lifted state peut être supprimé
- Dans le hook useFormations, exporter une nouvelle variable *memoize* correspondant aux statistiques par type de formation : type de formation, nombre de formation, note moyenne, charge heure restante cumulée
- Afficher cette liste dans un tableau en dessous du premier
- Créer un bouton refresh en amont des 2 tableaux et à chaque clic doit rafraichir les données (pour tester, modifier le fichier json en conséquence)

REACT

TP

- Ajouter un formulaire en dessous des tableaux comprenant des inputs relatifs aux propriétés d'une formation, et un bouton ajouter
- Au clic sur ajouter, enregistrer cette nouvelle formation dans le backend. Rafraichir en conséquence les tableaux
- Créer un context userContext dans un dossier context, qui récupère l'utilisateur 1 depuis le backend et qui le propage à l'ensemble du composant App. Dans le composant App, ajouter un Header et y afficher le nom de l'utilisateur du context
- Bonus : intégrer des spinners lors du refresh de la data dans le bouton « Refresh »

REACT

ROUTER

- **React Router** est une bibliothèque de routage pour React permettant la navigation
- Permet de gérer des routes déclaratives via des composants grâce à <Outlet>, <Link>, <NavLink>, <Routes>, <Route> et <BrowserRouter>
- Supporte la navigation dynamique, **les paramètres d'URL**, les routes imbriquées et les redirections
- Permet d'implémenter **des guards et des layouts spécifiques** pour certaines routes
- Supporte **la navigation programmée** via le hook
- Supporte la navigation en **chargement à la demande avec React.lazy**
- Fournit des hooks utiles comme **useParams, useMatch et useLocation**

REACT

ROUTER : INITIALISATION

- À la racine du projet, installer les dépendances :
  - *npm install react-router-dom*
  - *npm install -D @types/react-router-dom*
- Positionner le Router en contexte global :

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.tsx'
import { BrowserRouter } from 'react-router-dom'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </StrictMode>,
)
```

REACT

ROUTER : LA DÉCLARATION DE ROUTES

```jsx
import { Routes, Route, Navigate } from 'react-router-dom'
import { Home } from './pages/Home';
import { About } from './pages/About';
import { NotFound } from './pages/NotFound';
import { lazy, Suspense } from 'react'

const UserProfileLazy = lazy(() => import('./pages/UserProfile'))

export default function AppRoutes() {
  return (
    <Suspense fallback={<div>Chargement...</div>}>
      <Routes>
        {/* Route par défaut */}
        <Route path="/" element={<Home />} />

        {/* Redirection */}
        <Route path="/accueil" element={<Navigate to="/" replace />} />

        {/* Route classique */}
        <Route path="/about" element={<About />} />

        {/* Route lazy avec path param */}
        <Route path="/user/:id" element={<UserProfileLazy />} />

        {/* Route 404 */}
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Suspense>
  )
}
```

REACT

ROUTER : NESTED ROUTED

```jsx
<Route path="/dashboard" element={<DashboardLayout />}>
  <Route index element={<DashboardHome />} />
  <Route path="settings" element={<DashboardSettings />} />
</Route>
```

```jsx
import { Outlet } from "react-router-dom"

export default function DashboardLayout() {
  return (
    <div>
      <DashBoardHeader />
      <DashboardContent>
        <Outlet /> {/* C'est ici que les sous-routes s'affichent */}
      </DashboardContent>
    </div>
  )
}
```

REACT

ROUTER : LES LIENS DE NAVIGATION

```jsx
import React from 'react'
import { Link, NavLink } from 'react-router-dom'

export default function NavMenu() {
  return (
    <nav>
      {/* Link simple */}
      <Link to="/">Accueil</Link>

      {/* NavLink avec possibilité d'activer un style en fonction de l'état du router */}
      <NavLink to="/about" className={({ isActive }) => (isActive ? 'active' : undefined)}> À propos </NavLink>
    </nav>
  )
}
```

REACT

ROUTER : PATH ET QUERY PARAMS

```tsx
// /products/123?openExportMode=true

import { useParams, useLocation } from "react-router-dom"
import { useEffect } from "react"

export default function ProductDetails() {
  // Path param
  const { productId } = useParams<{ productId: string }>()

  // Query params
  const location = useLocation()
  const queryParams = new URLSearchParams(location.search)
  const openExportMode = queryParams.get("openExportMode")
```

REACT

ROUTER : TP

- Créer un composant FormationDetail dans un dossier pages/FormationDetail
  - Ce composant récupère depuis les path params l'id de la formation sur la route /formations/<id_formation>
  - Il récupère depuis le backend la formation correspondant à cette id et affiche l'ensemble de cette formation
- Créer un composant NotFound dans un dossier pages/NotFound et qui renvoie une div avec : Cette Page n'existe pas !
- Créer un composant About dans un dossier pages/About avec un lorem
- Positionner des routes à la racine de l'application, dans le composant App :
  - Une route racine qui redirige vers le composant FormationManager
  - Une route /formations/<id_formation> qui redirige vers le composant FormationDetail
  - Une route /about qui redirige en lazy loading vers le composant About
  - Une route /accueil qui redirige vers la racine
  - Une route joker qui redirige vers le composant NotFound

REACT

ROUTER : TP

- Dans le composant App, en amont des routes, créer un entête avec un menu de navigation :
  - Un lien vers la racine : Accueil
  - Un lien vers about : A propos

  Ces liens doivent s'activer en fonction de la navigation

- Dans le tableau du FormationManager, créer un lien pour chaque formation qui renvoie vers le détail associé

REACT

ROUTER : GUARDS

```jsx
import { Navigate, Outlet, useLocation } from "react-router-dom"

export default function RequireAuth() {
  const isAuthenticated = useAuth()
  const location = useLocation

  if (!isAuthenticated) {
    return <Navigate to="/login" state={{ from: location }} replace />
  }

  return <Outlet />
}
```

```jsx
<Route element={<RequireAuth />}>
  <Route path="/app">
    <Route index element={<Dashboard />} />
    <Route path="profile" element={<Profile />} />
  </Route>
</Route>
```

REACT

ROUTER : TP

- Créer un composant NotAuthorized dans un dossier pages/NotAuthorized et qui renvoie une div : Vous n'êtes pas autorisé à accéder à la ressource <url de la ressource>. Ce composant sera routé sur /not-authorized
- Créer un composant Auth dans un dossier layout, vérifiant l'habilitation du user : pour ce faire, récupérer de manière statique l'appel au backend /user/id et vérifier si le user a pour rôle administrateur
  - Si c'est le cas, il libère la navigation
  - Sinon il renvoie vers le composant NotAuthorized en indiquant la ressource demandée
- Positionner le composant Auth en amont de la route /formations/<id_formation>
- Modifier le json du backend pour tester

REACT

GESTION DES FORMULAIRES

Méthode	Avantages	Inconvénients	Cas d'usage recommandé
HTML natif	- Simple et rapide à mettre en place - Validation automatique par le navigateur	- Personnalisation limitée - Pas de contrôle dans React - Feedback peu personnalisable	- Formulaire très simple - Prototypes rapides
Validation manuelle en JS	- Contrôle total sur les règles et messages - Bonne intégration avec React - Sur mesure	- Beaucoup de code à écrire - Gestion d'état répétitive - Peu maintenable si complexe	- Formulaires petits à moyens - Besoins très spécifiques
Avec bibliothèque	- Code concis et clair - Haute performance (ex. react-hook-form) - Compatible avec des schémas (yup, zod) - Bon support TypeScript	- Nécessite une dépendance externe - Courbe d'apprentissage légère si débutant	- Formulaires moyens à complexes - Projets professionnels homogènes

REACT

GESTION DES FORMULAIRES : YUP

- **Librairie JavaScript** de validation par **schémas**
- Déclare des **règles** simples et composables
- Valide **chaînes**, **nombres**, **dates**, **objets**, **tableaux**
- Messages d’erreur personnalisables
- Supporte **validations conditionnelles**
- Compatible avec **React** et **TypeScript**

```
import * as yup from 'yup';

const schema = yup.object({
  email: yup.string().email('Email invalide').required('Email requis'),
  password: yup.string().min(6, 'Mot de passe trop court').required('Mot de passe requis'),
});
```

REACT

GESTION DES FORMULAIRES :USEFORM DU TEMPLATE

- Copier le hook useForm depuis le template et installer yup :
  - *npm install --save yup*

```
import * as yup from "yup";
import useForm from './hooks/useForm';

const validationSchema = yup.object({
  login: yup.string().required('Le login est requis').min(2, 'Le login fait au moins 2 caractères'),
  pwd: yup.string().required('Le password est requis').min(5, 'Le password fait au moins 5 caractères')
})

function Form() {

  const handleSubmit = (data: { login: string, pwd: string}, errors: unknown) => { console.log(data, errors) }

  const { formData: loginForm, errors, register }= useForm<{ login: string, pwd: string}>
    (handleSubmit, { login: '', pwd: '' }, validationSchema);

  return (
    <>
      <form>
        <div>
          <label>Login</label>
          <input type="text" value={loginForm.login} {...register('login')} />
        </div>
        <div>
          <label>Mot de passe</label>
          <input type="password" value={loginForm.pwd} {...register('pwd')} />
        </div>
        { errors && (
          <>
            <div>{errors.login}</div>
            <div>{errors.pwd}</div>
          </>
        )}
      </form>
    </>
  )
}
```

REACT

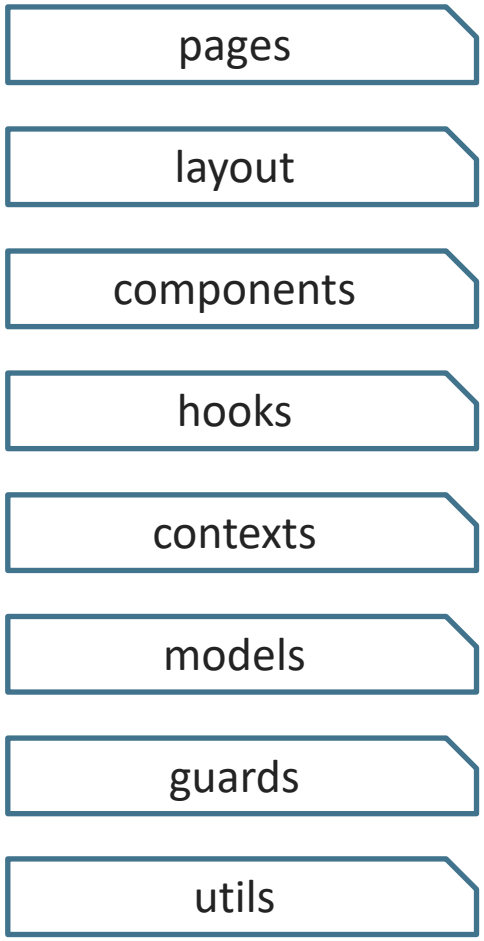
GESTION DES FORMULAIRES : TP

- En dessous des tableaux de formations, créer un formulaire comportant tous les champs d'une formation : 3 inputs pour nom, chargeH et note et 1 select pour le typeF
- Créer un object de validation définissant les règles suivantes :
  - nom: chaine de caractères, requis
  - chargeH: nombre, nombre requis
  - typeF: chaine de caractères, requis
  - note: nombre, minimum 0, maximum 10
- Ajouter un bouton à la suite du formulaire et une div. En cas d'erreur, afficher les erreurs dans la div et griser le bouton
- Au clic sur le bouton enregistrer cette nouvelle formation (penser à mettre à jour le tableau)

1. Présentation des outils
2. Javascript et Typescript
3. Écosystème
4. React
5. **Architecture logicielle**

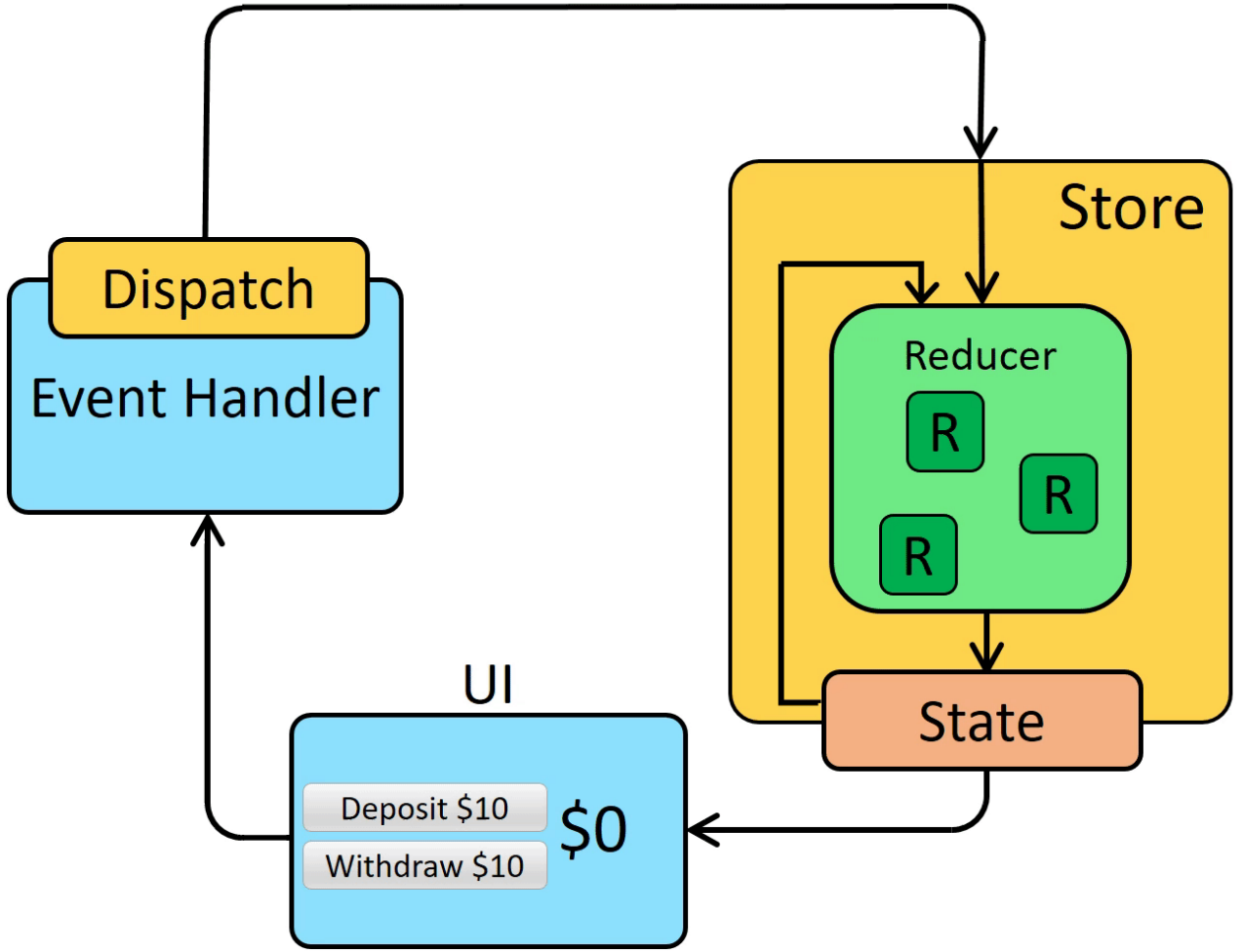
ARCHITECTURE LOGICIELLE

STRUCTURE



ARCHITECTURE LOGICIELLE

REDUX



ARCHITECTURE LOGICIELLE

REDUX : RTK

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

export const fetchInitialCount = createAsyncThunk(
  'counter/fetchInitialCount',
  async () => {
    const res = await fetch('/api/initial-count');
    const data = await res.json();
    return data.value;
  }
);

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
    status: 'idle',
    error: null,
  },
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 },
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchInitialCount.pending, (state) => {
        state.status = 'loading';
      })
      .addCase(fetchInitialCount.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.value = action.payload;
      })
      .addCase(fetchInitialCount.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.error.message;
      });
  },
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { store } from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

ARCHITECTURE LOGICIELLE

REDUX : RTK

```jsx
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchInitialCount, increment, decrement } from './counterSlice';

function App() {
  const count = useSelector((state) => state.counter.value);
  const status = useSelector((state) => state.counter.status);
  const error = useSelector((state) => state.counter.error);
  const dispatch = useDispatch();

  useEffect(() => {
    dispatch(fetchInitialCount());
  }, [dispatch]);

  if (status === 'loading') return <p>Loading...</p>;
  if (status === 'failed') return <p>Error: {error}</p>;

  return (
    <div>
      <h1>Count from API: {count}</h1>
      <button onClick={() => dispatch(decrement())}>-</button>
      <button onClick={() => dispatch(increment())}>+</button>
    </div>
  );
}

export default App;
```

ARCHITECTURE LOGICIELLE

TESTS AUTOMATISÉS

- Utilise souvent **Jest** pour l'environnement de test (peut utiliser **Vitest**)
- Utilise **React Testing Library (@testing-library/react)** pour interagir avec le DOM.
- Teste des **rendus**, **interactions utilisateur**, **états**, **props**, **callbacks**, etc.
- Doit rester **isolé**, **rapide** et **prévisible**.
- Doit tester **le parcours utilisateur** et non pas la vision développeur

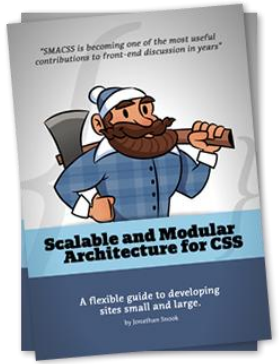
ARCHITECTURE LOGICIELLE

GESTION DES STYLES

Web Components

Component

Feuille de style



SMACSS

Base

Layout

Module

State

Theme

Variables

ARCHITECTURE LOGICIELLE

ESLINT

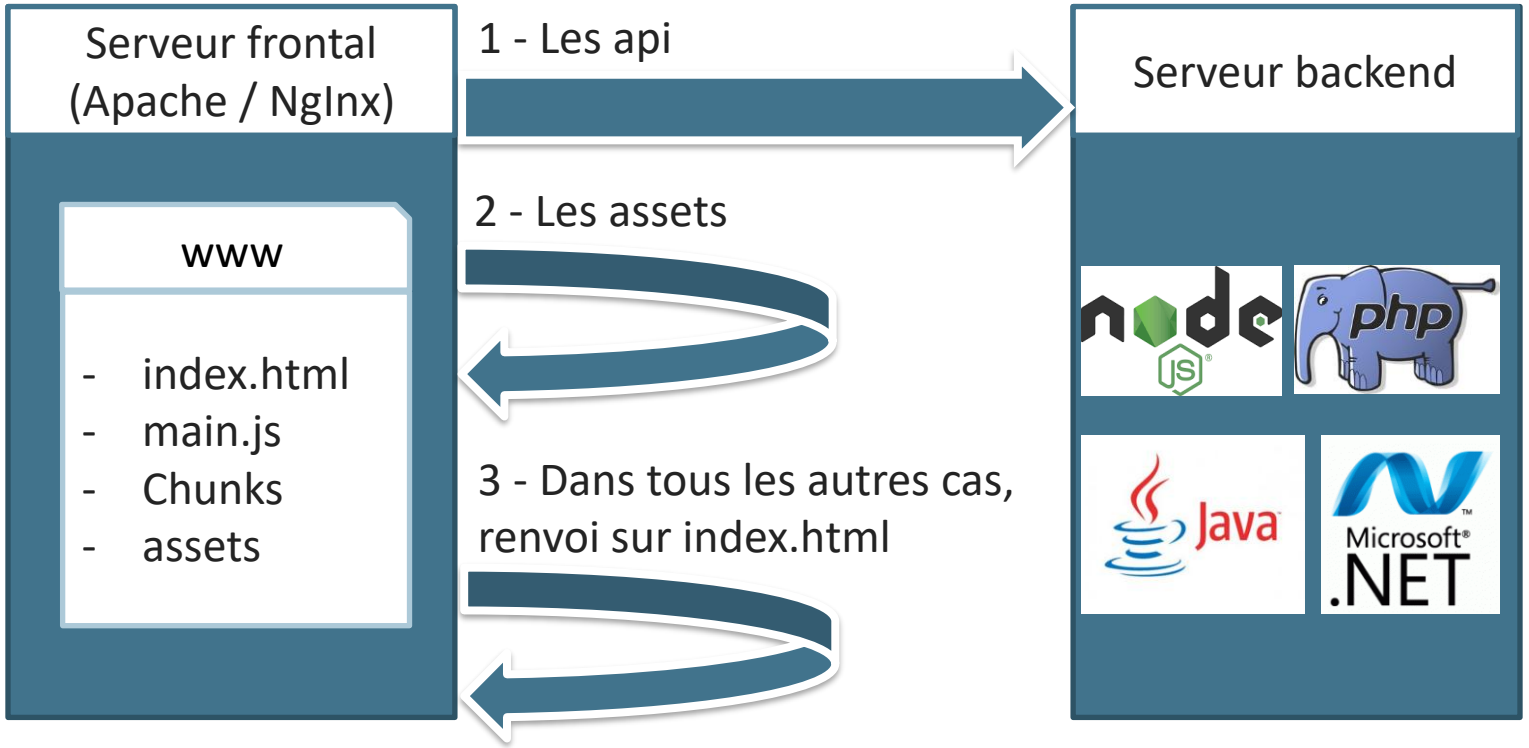
- Vite et Next se base sur la librairie ES Lint pour l'analyse statique du code
- Permet de configurer les règles :
  - Formatage et indentation
  - Règles de développement associées au langage JavaScript
  - Règles de développement associées au langage TypeScript
  - Règles de développement associées à React

ARCHITECTURE LOGICIELLE

DEBUGGING

ARCHITECTURE LOGICIELLE

LE DÉPLOIEMENT



Routage et affichage de données

Formulaire, récupération et envoi de données

Affichage de graphiques avec une librairie

Carte blanche

TP 1

ROUTAGE ET AFFICHAGE DE DONNÉES

- Créer un fichier json avec 3 jeux de données (e.g. les personnages, les arènes et les compétences)
  - Les éléments doivent se référencer les uns les autres (dans l'exemple, un personage a une arène de prédilection et une liste de compétences)
- Servir le fichier json au travers d'un json-server
- Créer une application avec 3 pages de navigation correspondant à chaque jeu de données. Dans chacune de ces pages, il est possible de voir l'ensemble des éléments du jeu de données dans une liste non ordonnée
- Chaque élément d'une liste est un lien qui peut rediriger vers une page de detail. Chaque élément référencé (e.g. l'arène de prédilection est cliquable et permet de voir le détail de cette arène)

TP 2

FORMULAIRE

- Créer un formulaire avec plusieurs champs et gérer les critères de validation suivants :
  - Un champ qui respecte une expression régulière (e.g. un numéro de sécurité sociale)
  - Un champ de type email
  - Un champ de type date qui n'accepte que des dates dans le future
- Afficher les messages de validation au submit et lorsque l'utilisateur qui un champs
- Pour chaque champs en erreur, le colorizer en rouge afficher son message juste en dessous
- Comment factoriser ça dans un composant dédié ?

TP 3

INTÉGRATION D'UNE LIBRAIRIE

- Intégrer la libraire Chartjs
- Créer un fichier json qui représentera un jeu de données (traces, valeur / temps, …). Servir ce fichier par json-server
- Afficher un graphique de ces données (type chart)
- Afficher un camembert de ces données (type PieChart)

QUESTIONS / RÉPONSES

RESSOURCES

- Documentation officielle : https://react.dev/reference/react
- Les Hooks natifs : https://react.dev/reference/react/hooks
- Composants natifs : https://react.dev/reference/react/components
- Documentation de Vite : https://vite.dev/guide/
- Documentation de NextJs : https://nextjs.org/docs
- Documentation de RTK : https://redux-toolkit.js.org/usage/usage-guide
- SMACSS : https://smacss.com/