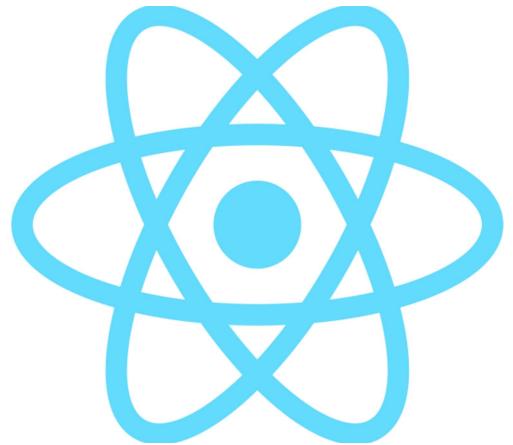


ReactJS



Créer une appli. React avec *Vite*



- Créer le projet:

```
npm create vite@latest
```

- Pourquoi utiliser '*vite*' ?

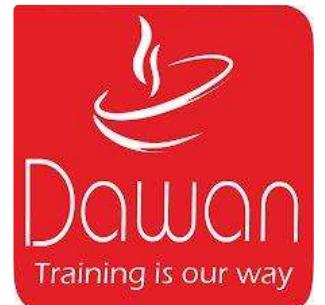
- Structures de base d'un projet react créé automatiquement
- Compilation rapide
- Rechargement à chaud (les modifications que vous apportez à votre code sont immédiatement reflétées dans le navigateur)

Qu'est-ce que React ?



- React est une **bibliothèque** JavaScript déclarative, efficace et flexible pour construire des interfaces utilisateurs (UI). Elle vous permet de composer des UI complexes à partir de petits morceaux de code isolés appelés « **composants** ».

Pourquoi utiliser React ?

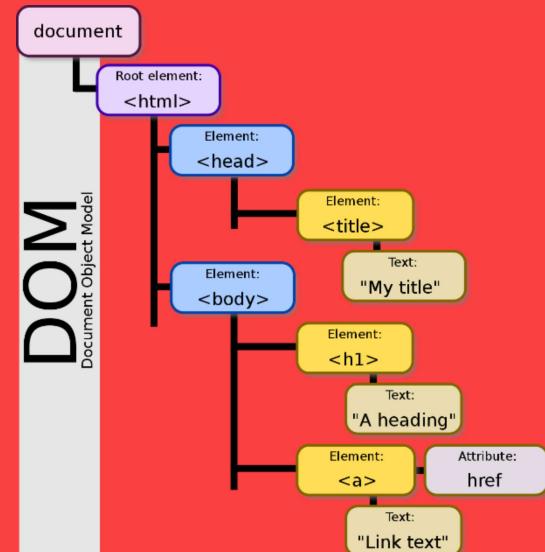


- **Virtual DOM:** React utilise un concept appelé Virtual DOM (Document Object Model) qui permet de mettre à jour efficacement l'interface utilisateur. Plutôt que de manipuler directement le DOM, React crée une représentation virtuelle de celui-ci en mémoire, ce qui permet des mises à jour plus rapides et efficaces.
- **Composants réutilisables:** React encourage le développement d'interfaces utilisateur modulaires à l'aide de composants. Les composants peuvent être réutilisés à différents endroits de l'application, ce qui favorise la maintenabilité et la scalabilité du code.
- **JSX:** React utilise JSX (JavaScript XML), une extension de syntaxe qui permet d'écrire du code HTML à l'intérieur de JavaScript. Cela rend le code plus lisible et plus facile à écrire, en permettant aux développeurs de décrire l'apparence de l'interface utilisateur directement dans le code JavaScript.
- **Unidirectional Data Flow:** React suit le principe de flux de données unidirectionnel, ce qui signifie que les données se déplacent d'un composant parent à ses composants enfants. Cela rend le code plus prévisible et plus facile à comprendre, en facilitant le suivi des données à travers l'application.
- **Large écosystème et communauté active:** React est largement adopté par la communauté des développeurs, ce qui se traduit par un vaste écosystème d'outils, de bibliothèques et de ressources disponibles. La communauté est également très active, ce qui signifie qu'il est facile de trouver de l'aide, des tutoriels et des exemples de code.
- **Performance:** Grâce à son utilisation du Virtual DOM et à son approche de rendu efficace, React offre généralement de bonnes performances, même pour les applications web complexes.
- **Support de la part de Facebook:** React a été développé et est maintenu par Facebook, ce qui lui confère une solide base de support et de développement. De nombreuses grandes entreprises utilisent React dans leurs produits, ce qui garantit une certaine pérennité et stabilité à la technologie.

Le DOM (Document Object Model)



- Le DOM est une interface de programmation (API) utilisée dans les langages de script, principalement JavaScript, pour représenter et manipuler la structure, le contenu et le style des documents HTML, XML et XHTML
- Le DOM fournit une représentation hiérarchique et structurée de la page web chargée dans le navigateur. Cette représentation est organisée sous forme d'un arbre où chaque élément HTML (comme les balises `<div>`, `<p>`, ``, etc.) est représenté par un nœud dans cet arbre.



React: le virtual Dom



Les étapes :

1. **Création d'une représentation virtuelle du DOM** : Lorsque vous créez des composants React et que vous définissez leur structure en JSX, React ne manipule pas directement le DOM du navigateur. Au lieu de cela, il crée une représentation virtuelle de cet arbre DOM dans la mémoire de l'ordinateur.
2. **Comparaison avec le DOM réel** : À chaque modification de l'état d'un composant (par exemple, lorsqu'une donnée change ou lorsqu'un événement se produit), React recrée cette représentation virtuelle de l'arbre DOM. Il compare ensuite cette nouvelle représentation virtuelle avec l'ancienne.
3. **Détermination des changements nécessaires** : React détermine les différences entre la nouvelle représentation virtuelle et l'ancienne en utilisant un algorithme de différenciation efficace. Cet algorithme compare les deux arbres de manière itérative et identifie les nœuds qui ont changé.
4. **Mise à jour sélective du DOM réel** : Une fois que les changements ont été déterminés, React met à jour sélectivement le DOM réel du navigateur pour refléter ces changements. **Plutôt que de mettre à jour tout l'arbre DOM, React ne met à jour que les parties qui ont effectivement changé**.
5. **Rendu dans le DOM réel** : Enfin, React effectue le rendu des modifications dans le DOM réel du navigateur, ce qui permet à l'interface utilisateur de refléter l'état actuel de l'application.

Le JSX



- JSX est une extension de syntaxe pour JavaScript qui vous permet d'écrire un balisage de type HTML dans un fichier JavaScript.
- Le Web a été construit sur HTML, CSS et JavaScript. Pendant de nombreuses années, les développeurs Web ont conservé le contenu en HTML, la conception en CSS et la logique en JavaScript, souvent dans des fichiers séparés.
- Mais à mesure que le Web devenait plus interactif, la logique déterminait de plus en plus le contenu. JavaScript était en charge du HTML ! **C'est pourquoi dans React, la logique de rendu et le balisage cohabitent au même endroit : les composants.**

Le JSX



- JSX produit des « éléments » React.

```
const element = <h1>Bonjour, monde !</h1>;
```

- **Chaque composant React est une fonction JavaScript qui peut contenir des balises que React affiche dans le navigateur.**
- JSX ressemble beaucoup à HTML, mais il est un peu plus strict et peut afficher des informations dynamiques

Le JSX



- JSX n'est rien d'autre qu'une expression:
 - Après la compilation, les expressions JSX deviennent de simples appels de fonctions JavaScript
 - Ça signifie que vous pouvez utiliser JSX à l'intérieur d'instructions if ou de boucles for, l'affecter à des variables, l'accepter en tant qu'argument, et le renvoyer depuis des fonctions
- Il est possible d'utiliser des accolades pour utiliser une expression JavaScript dans un attribut :

```
const element = <img src={user.avatarUrl}></img>;
```

Le JSX



- Dans la mesure où JSX est plus proche de JavaScript que de HTML, React DOM utilise la casse camelCase comme convention de nommage des propriétés, au lieu des noms d'attributs HTML.
- Par exemple, '*class*' devient '**className**' en JSX, et '*tabindex*' devient '**tabIndex**'.
- JSX empêche les attaques d'injection : **Vous ne risquez rien en utilisant une saisie utilisateur dans JSX.**
 - Par défaut, **React DOM échappe toutes les valeurs intégrées avec JSX** avant d'en faire le rendu. Il garantit ainsi que vous ne risquez jamais d'injecter quoi que ce soit d'autre que ce que vous avez explicitement écrit dans votre application. **Tout est converti en chaîne de caractères avant de produire le rendu.** Ça aide à éviter les attaques XSS (cross-site-scripting)

Le JSX



- Babel* compile JSX vers des appels à **React.createElement()**. Avec Babel, les deux exemples ci-dessous sont donc identiques:

```
const element = (
  <h1
    className="greeting">
    Bonjour, monde !
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Bonjour, monde !'
);
```

// Remarque : cette structure est simplifiée

```
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Bonjour, monde !'
}
};
```

- **React.createElement()** effectue quelques vérifications pour vous aider à écrire un code sans bug, mais pour l'essentiel il crée un objet qui ressemble à ceci :

Babel: outil de compilation qui permet de transformer le code JSX utilisé avec React en code JavaScript standard, ce qui permet à React de fonctionner dans les navigateurs.

Présentation de la structure d'un projet react



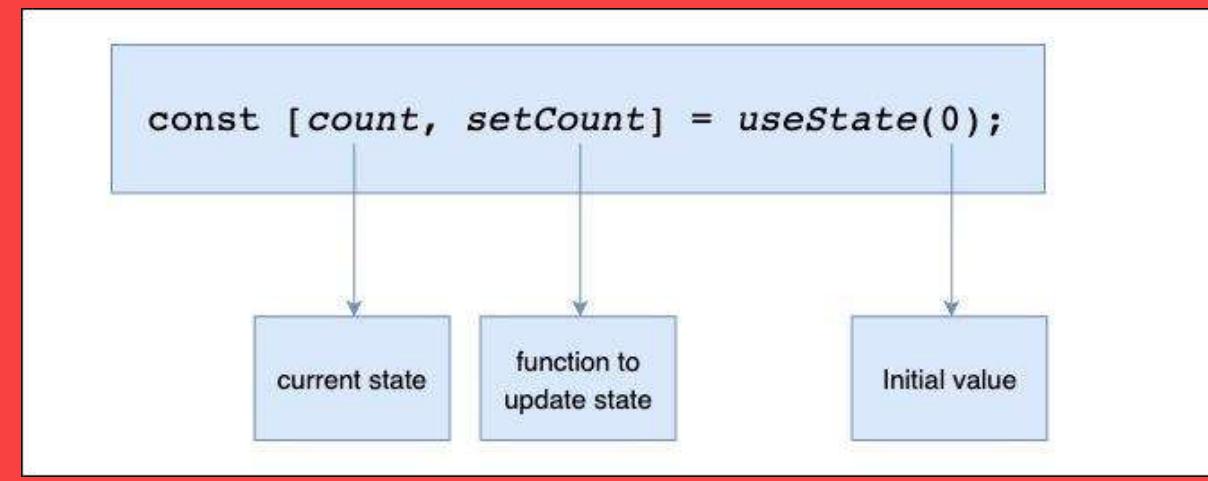
```
my-react-app/
├── public/
│   └── index.html
│   ...
├── src/
│   ├── components/
│   │   ├── Header/
│   │   │   ├── Header.jsx
│   │   │   ├── Header.css
│   │   │   ...
│   │   ├── Footer/
│   │   │   ├── Footer.jsx
│   │   │   ├── Footer.css
│   │   │   ...
│   │   ...
│   └── pages/ (ou/view)
│       └── Home/
```

- **public/** : Ce dossier contient les fichiers statiques de l'application qui seront servis tels quels par le serveur, comme le fichier HTML principal (`index.html`), les images, les polices, etc.
- **src/** : Ce dossier contient tout le code source de l'application.
 - **components/** : Ce dossier contient les composants réutilisables de l'application, organisés par fonctionnalité ou par type.
 - **pages/** : Ce dossier contient les composants de niveau supérieur qui représentent les différentes pages de l'application
 - **App.js** : Le composant racine de l'application, qui contient la structure de base de l'interface utilisateur et définit les routes principales de l'application
 - **index.js** : Le point d'entrée principal de l'application, où React est généralement rendu dans le DOM
- **hooks/** : hooks personnalisés réutilisables. Les hooks sont des fonctions JavaScript qui permettent de partager de la logique entre les composants de manière plus modulaire
- **package.json** : Le fichier de configuration de Node.js qui spécifie les dépendances du projet, les scripts npm, etc.
- **README.md** : Un fichier contenant des informations importantes sur le projet, telles que des instructions d'installation, d'utilisation, etc.

Le hook useState



Ce hook permet à un composant fonctionnel de conserver un état local **et de déclencher des ré-renders lorsque cet état change**



Flux de données



- **De parents à enfants:**

- -> Via les props

Cela se fait généralement en passant des données via les props. Le parent peut passer des données aux enfants en leur fournissant des props, qui sont essentiellement des attributs d'objet JavaScript. Les enfants peuvent ensuite utiliser ces props pour afficher des informations ou effectuer des actions en fonction des données reçues.

- **D'enfants vers parents:**

- -> via des événements

Dans ce cas, les enfants peuvent envoyer des données aux parents en déclenchant des événements. Les parents peuvent écouter ces événements et effectuer des actions en réponse. Pour cela, les enfants peuvent utiliser des fonctions de rappel (callbacks) passées en tant que props par les parents, ou utiliser le 'context' dans les cas où plusieurs niveaux de profondeur les séparent.

Les class



- Dans un composant de classe, le cycle de vie se compose de plusieurs étapes clés:

1. Mounting (Montage) :

- constructor(): Le composant est initialisé.
- componentWillMount() (obsolète): Appelé juste avant que le composant ne soit rendu pour la première fois.
- render(): Rendu du composant dans le DOM.
- componentDidMount(): Appelé juste après que le composant est monté dans le DOM. C'est un bon endroit pour effectuer des appels réseau ou initialiser des données.

2. Updating (Mise à jour) :

- componentWillReceiveProps() (obsolète): Appelé lorsque le composant reçoit de nouvelles props.
- shouldComponentUpdate(): Permet de contrôler si le composant doit être mis à jour ou non.
- componentWillUpdate() (obsolète): Appelé juste avant la mise à jour du composant.
- render(): Le composant est à nouveau rendu.
- componentDidUpdate(): Appelé après la mise à jour du composant.

3. Unmounting (Démontage) :

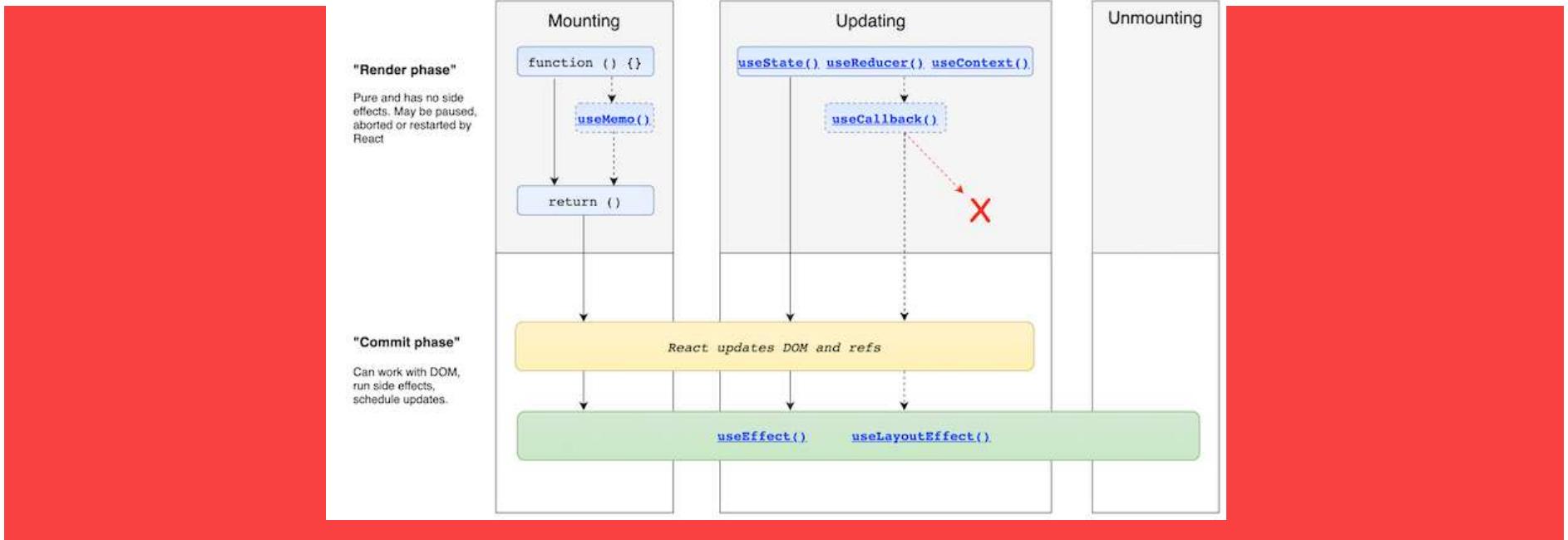
- componentWillUnmount(): Appelé juste avant que le composant soit retiré du DOM. C'est l'endroit idéal pour nettoyer les ressources telles que les abonnements ou les timers.



Cycle de vie



React Hooks Lifecycle



Les formulaires



Les formulaires classiques:

```
<form onSubmit={handleSubmit2}>
  <input
    type="text"
    name="login"
    defaultValue="login"
    className="mb-5 ml-5 input input-bordered"
  />
  <input type="text" name="password" defaultValue="password" />
  <button className="ml-2 btn">Envoyer</button>
</form>
```

Dans cet exemple, la fonction exécutée au moment de la validation des '*handleSubmit2*'

Le hook useEffect



- Il permet d'effectuer des effets secondaires dans les composants fonctionnels
- Le *hook* `useEffect` est le **couteau suisse de tous les hooks disponibles**. C'est la meilleure solution à beaucoup de bug tel que :
 - Comment exécuter du code lorsqu'un *state* ou une *prop* change
 - Comment configurer des *timers* ou des intervalles
 - Comment récupérer des données lors du montage d'un composant
 -

Le hook useEffect



```
useEffect(setup, dependencies?)
```

- Appelez `useEffect` au niveau supérieur de votre composant pour déclarer un effet
- Lorsque votre composant est ajouté au DOM, React exécutera votre fonction de configuration. Après chaque rendu avec des dépendances modifiées, React exécutera d'abord la fonction de nettoyage (si vous l'avez fournie) avec les anciennes valeurs, puis exécutera votre fonction de configuration avec les nouvelles valeurs.

Le hook useEffect



useEffect(setup, dependencies?)

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

Le hook useMemo



- Hook de gestion d'état dans React, **utilisé pour optimiser les performances en mémorisant la valeur calculée entre les rendus**, afin d'éviter des calculs inutiles
- En React, chaque fois que l'état d'un composant change ou que celui-ci est rendu à nouveau, **le rendu complet est recalculé**, y compris les calculs de valeurs dérivées. Cela peut être inefficace lorsque les calculs sont coûteux en ressources et que la valeur calculée reste la même entre les rendus successifs
 - **useMemo permet de résoudre ce problème en mémorisant la valeur calculée**, et en ne recalculant cette valeur que si les dépendances fournies à useMemo changent.

useMemo VS useEffect



- Dans certains cas, vous pouvez obtenir un comportement similaire à celui de useMemo en utilisant useEffect
- Les différences importantes entre les deux sont:
 - useMemo est conçu spécifiquement pour la mémorisation de valeurs calculées, tandis que useEffect est destiné à la gestion des effets secondaires.
 - useEffect entraînera toujours un rendu supplémentaire lorsqu'il est exécuté, car il modifie l'état du composant. useMemo, en revanche, ne provoque pas de rendu supplémentaire lorsqu'il renvoie la valeur mémorisée
 - Utiliser useMemo lorsque vous avez l'intention de mémoriser une valeur calculée **rend votre intention plus claire pour d'autres développeurs** qui liront votre code.

React hook form



- React Hook Form est une bibliothèque légère et performante pour la gestion des formulaires dans les applications React. Voici un résumé rapide de ses fonctionnalités et de son utilisation :
 - **Basé sur les hooks** : React Hook Form exploite les fonctionnalités des hooks de React pour gérer l'état des formulaires et leurs validations.
 - **Optimisé pour les performances** : La bibliothèque est conçue pour être rapide et légère, en minimisant les rendus inutiles et en évitant les re-calcus coûteux
 - **Validation flexible** : Elle offre une validation flexible et performante des formulaires, **avec des règles de validation personnalisées**, des messages d'erreur et une gestion des états de validation.
 - **Contrôle des champs** : Elle permet de contrôler facilement les champs de formulaire, en récupérant les valeurs saisies par l'utilisateur et en gérant leur état
 - **Support pour les formulaires complexes** : Elle prend en charge les formulaires complexes avec des champs imbriqués, des tableaux de champs et d'autres structures de données complexes.
 - **Extensible et personnalisable** : Elle permet d'étendre ses fonctionnalités et de personnaliser son comportement selon les besoins spécifiques de l'application.



React hook form

```
</> useForm
  - </> register
  - </> unregister
  - </> formState
  - </> watch
  - </> handleSubmit
  - </> reset
  - </> resetField
  - </> setError
  - </> clearErrors
  - </> setValue
  - </> setFocus
  - </> getValues
  - </> getFieldState
  - </> trigger
  - </> control
  - </> Form
```

Le hook useForm retourne un objet contenant des méthodes et des propriétés utiles pour la gestion du formulaire.

on peut notamment noter:

- Register: <https://react-hook-form.com/docs/useform/register>
- HandleSubmit: <https://react-hook-form.com/docs/useform/handlesubmit>
- FormState: <https://react-hook-form.com/docs/useform/formstate>
- SetValue: <https://react-hook-form.com/docs/useform/setvalue>
- ClearErrors: <https://react-hook-form.com/docs/useform/clearerrors>

React router dom



- React Router DOM est une bibliothèque populaire pour la navigation dans les applications React
- **Gestion de la navigation** : React Router DOM permet de gérer la navigation entre différentes vues ou pages au sein d'une application React, sans avoir besoin de recharger la page entière.
- **Déclaration des routes** : Il offre des composants tels que `<BrowserRouter>`, `<HashRouter>`, `<Route>`, `<Switch>`, etc., pour définir les routes de l'application et associer des composants à des URL spécifiques.
- **Routing déclaratif** : Les routes sont définies de manière déclarative, ce qui facilite la compréhension et la maintenance du code. Les composants sont rendus en fonction de l'URL actuelle.
- **Passage de paramètres d'URL** : Il permet de passer des paramètres d'URL aux composants, permettant ainsi de créer des routes dynamiques qui affichent des données en fonction de l'URL.
- **Gestion des paramètres de requête** : Il prend en charge la gestion des paramètres de requête dans les URL, facilitant ainsi la construction d'URL avec des paramètres et leur récupération dans les composants.
- **Gestion de l'historique de navigation** : React Router DOM offre un objet `history` pour naviguer vers des URL précédentes, avancer ou reculer dans l'historique de navigation, et effectuer d'autres actions de navigation programmatiquement.
- **Intégration avec React** : Il s'intègre de manière transparente avec d'autres fonctionnalités de React, ce qui permet d'utiliser des hooks tels que `useHistory`, `useParams`, `useLocation`, etc., pour accéder aux fonctionnalités de navigation.

ESLINT



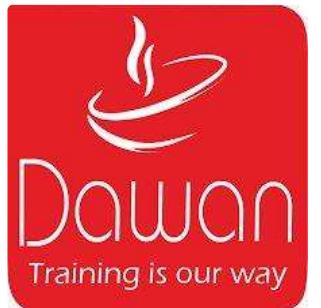
- **Qu'est-ce qu'ESLint ?**

ESLint est un outil d'analyse de code statique pour JavaScript (et ses dérivés comme TypeScript). Son objectif est de détecter et signaler les problèmes de qualité du code et les erreurs potentielles, tout en appliquant des règles de style personnalisables.

En React, ESLint est un outil clé pour :

- Maintenir une cohérence dans le style de codage.
- Prévenir les erreurs courantes de développement.
- Encourager les bonnes pratiques comme l'échappement des textes et la validation des props.

ESLINT: Résumé des objectifs



- **Qualité du code** : Maintenir un code propre et sans erreurs avec les règles recommandées pour JavaScript et React.
- **Sécurité** : Renforcer la sécurité en React avec des règles comme celles des hooks ou du JSX.
- **Performance** : Optimiser le flux de développement avec React Refresh (utile en mode développement).
- **Personnalisation** : Permet des ajustements pour des cas spécifiques (`jsx-no-target-blank` désactivée).



ESLINT: respect des règles définies

- Que se passe-t-il si je ne respecte pas des règles définies

1. Avertissement (warn)

Si une règle est configurée avec un niveau de严重性 de **warn**, ESLint affichera un **avertissement** dans la console lorsque la règle est enfreinte.

```
// Avertir si des exports non liés à des composants React sont utilisés.  
// Autoriser les exports constants grâce à l'option `allowConstantExport`.  
"react-refresh/only-export-components": [  
  "warn",  
  { allowConstantExport: true },  
],
```

Conséquences : Peut être ignoré temporairement, mais cela peut indiquer un problème potentiel dans le code

ESLINT: respect des règles définies



- Que se passe-t-il si je ne respecte pas des règles définies

2. Erreur (error)

- Si une règle est configurée avec un niveau de严重性 **error**, ESLint considérera l'infraction comme une **erreur bloquante**.
- Dans les environnements stricts (comme un CI/CD ou un build), cela **peut empêcher la compilation ou le déploiement du code**.

Conséquences : Le processus de développement ou de build sera interrompu jusqu'à ce que l'erreur soit corrigée.

ESLINT: respect des règles définies



- Vite, par défaut, ne configure pas ESLint pour qu'il soit strictement bloquant.

- **Avec une configuration par défaut d'ESLint intégrée à Vite :**

Les warnings apparaissent dans la console (généralement en jaune dans les éditeurs comme VSCode ou dans les logs de développement). Les erreurs peuvent aussi s'afficher, mais elles n'arrêtent pas le build par défaut.

- **A savoir ...**

ESLint est souvent utilisé pour aider les développeurs à écrire un code propre, mais son exécution en tant qu'étape bloquante nécessite des outils comme :

vite-plugin-eslint pour afficher les erreurs dans le navigateur.

Une intégration avec des hooks Git (lint-staged).

Une configuration CI/CD personnalisée.

Importation des modules

```
import js from '@eslint/js'  
import globals from 'globals'  
import react from 'eslint-plugin-react'  
import reactHooks from 'eslint-plugin-react-hooks'  
import reactRefresh from 'eslint-plugin-react-refresh'
```



- **@eslint/js**: Fournit les règles JavaScript de base recommandées par ESLint.
- **globals** : Contient des variables globales spécifiques, ici pour les environnements navigateur.
- **eslint-plugin-react** : Ajoute des règles spécifiques à React pour détecter des erreurs dans les composants ou les JSX.
- **eslint-plugin-react-hooks** : Assure le respect des règles des hooks (par exemple, utiliser useEffect correctement).
- **eslint-plugin-react-refresh** : Ajoute des vérifications liées au module de rafraîchissement à chaud (Hot Module Replacement - HMR).

ESLINT: Configuration par défaut du projet vite

```
export default [
  // Ignorer le dossier `dist` lors de l'analyse ESLint, car il contient des fichiers générés.
  { ignores: ["dist"] },

  {
    // Appliquer ESLint uniquement aux fichiers avec les extensions '.js' et '.jsx'.
    files: ["**/*.{js,jsx}"],

    // Définir les options pour l'analyse du langage JavaScript.
    languageOptions: {
      // Utiliser la version ECMAScript 2020 (qui inclut des fonctionnalités comme 'optional chaining').
      ecmaVersion: 2020,

      // Définir les variables globales de l'environnement navigateur (par exemple, 'window', 'document').
      globals: globals.browser,
    },
    parserOptions: {
      // Utiliser la version ECMAScript la plus récente disponible.
      ecmaVersion: "latest",

      // Activer la prise en charge du JSX.
      ecmaFeatures: { jsx: true },
    },
    sourceType: "module",
  },
  // Spécifier la version de React utilisée pour adapter les règles.
  settings: { react: { version: "18.3" } },
  // Ajouter des plugins spécifiques pour gérer React et ses fonctionnalités associées.
  plugins: {
    // Plugin React pour détecter les erreurs et appliquer les bonnes pratiques.
    react,
    // Plugin pour vérifier que les hooks React (comme 'useEffect') sont utilisés correctement.
    "react-hooks": reactHooks,
    // Plugin pour gérer les vérifications liées au rafraîchissement à chaud (React Refresh).
    "react-refresh": reactRefresh,
  },
]
```

```
// Définir les règles à appliquer.
rules: [
  // Règles de base recommandées par ESLint pour JavaScript.
  ...js.configs.recommended.rules,
  // Règles recommandées pour React (par exemple, nommage correct des composants, JSX valide).
  ...react.configs.recommended.rules,
  // Règles spécifiques pour le nouveau runtime JSX introduit dans React 17+.
  ...react.configs["jsx-runtime"].rules,
  // Règles pour s'assurer de l'utilisation correcte des hooks React.
  ...reactHooks.configs.recommended.rules,
  // Désactiver l'avertissement sur l'absence de `rel="noopener noreferrer"` avec `target="_blank"`.
  "react/jsx-no-target-blank": "off",
  // Avertir si des exports non liés à des composants React sont utilisés.
  // Autoriser les exports constants grâce à l'option `allowConstantExport`.
  "react-refresh/only-export-components": [
    "warn",
    { allowConstantExport: true },
  ],
]
```



ERROR ELEMENT ET ERROR BOUNDARY

ErrorBoundary: gérer les erreurs dans les composants



- Un ErrorBoundary est un composant qui intercepte les erreurs JavaScript dans ses enfants, permettant à l'application de ne pas planter en cas d'exception. Cela offre un moyen plus global de capturer des erreurs au niveau des composants.



Avant React router dom v6

Avant React Router v6, pour gérer les erreurs dans vos composants enfants ou dans vos routes, on utilisait une approche avec des Error Boundaries. Cela impliquait de créer un composant qui encapsule d'autres composants pour intercepter les erreurs JavaScript qui pourraient survenir. L'idée était de capturer les erreurs dans les composants enfants et d'afficher une interface alternative (comme un message d'erreur) sans que l'application ne plante.

Après v6: errorElement



Avec React Router v6, la gestion des erreurs au niveau du routage a été simplifiée grâce à la nouvelle propriété `errorElement`. Cette propriété permet de définir un composant qui sera affiché lorsqu'une erreur se produit dans une route spécifique, éliminant ainsi le besoin de recourir à un `ErrorBoundary` pour chaque route.

- **Ce que fait `errorElement` :**
 - **Gestion des erreurs spécifiques aux routes** : Si une erreur se produit dans la route spécifiée (par exemple, dans le rendu d'un composant associé à cette route), le `errorElement` est utilisé pour afficher un message d'erreur ou un autre composant d'erreur.
 - **Approche plus ciblée** : Au lieu de capturer les erreurs globalement avec un `ErrorBoundary`, React Router v6 permet de définir des comportements spécifiques pour chaque route via `errorElement`, rendant la gestion des erreurs plus modulaire et facile à appliquer.

En résumé



Conclusion

- Avant React Router v6, vous deviez utiliser un ErrorBoundary pour gérer les erreurs dans vos composants ou routes. Cela vous permettait de capturer les erreurs et d'afficher un message d'erreur ou une interface alternative.
- Avec la version 6 de React Router, la gestion des erreurs au niveau du routage est simplifiée grâce à la propriété errorElement. Cette approche est suffisante pour gérer les erreurs spécifiques aux routes sans avoir à utiliser un ErrorBoundary global pour chaque route ou composant.

Je n'ai plus besoin de page 404 ?



L'errorElement dans React Router est utilisé pour gérer des erreurs liées aux composants d'une route. Cela peut être utilisé pour afficher une page d'erreur générique ou spécifique lorsqu'une erreur survient dans la logique de votre composant.

Cependant, cela ne remplace pas directement la gestion des routes inexistantes

Dans React Router, pour gérer spécifiquement les URLs qui ne correspondent à aucune route définie, vous devez toujours définir un itinéraire avec un path: '*'.

Qu'est-ce que l'accessibilité ?

- Définition : Rendre les applications utilisables par tous, y compris les personnes en situation de handicap.
- Types de handicaps : moteur, visuel, auditif, cognitif.
- Importance :
 - Conformité légale (WCAG, RGAA).
 - Expérience utilisateur (UX) améliorée.
 - Valeur éthique.

Outils pour tester l'accessibilité :

- Navigateurs : Axe DevTools, Lighthouse.
- Navigateurs vocaux : NVDA, VoiceOver.
- Extensions Chrome : Accessibility Insights
-

1. Fournir des alternatives textuelles

- Les alternatives textuelles permettent aux lecteurs d'écran et autres technologies d'assistance de transmettre l'information des éléments non textuels.

- Images :

```
html
```

 Copier le code

```

```

2. Permettre une navigation clavier

Les utilisateurs doivent pouvoir naviguer sans souris, uniquement avec le clavier ou des dispositifs alternatifs (ex. : interrupteurs).

- Points clés :
 - Tous les éléments interactifs doivent être accessibles via la touche Tab.
 - Mettre le focus visuellement évident (CSS).

css

```
button:focus {  
    outline: 2px solid #005fcc;  
}
```

3. Garantir une bonne lisibilité

La lisibilité est essentielle pour les utilisateurs ayant des troubles visuels ou cognitifs.

- Aspects importants :

Contraste : Vérifiez un ratio minimum de 4.5:1 (pour le texte normal).

4. Fournir des messages d'erreur clairs

Les messages d'erreur doivent être compréhensibles et aidants, surtout pour les formulaires.

Bonnes pratiques :

- Indiquer ce qui est incorrect et comment le corriger.
- Associer les messages d'erreur à leurs champs avec `aria-describedby`.

```
<label for="email">Email :</label>
<input type="email" id="email" aria-describedby="email-error">
<span id="email-error" style="color: red;">Entrez une adresse email valide.</span>
```

5. Gérer les changements de contexte

Exemple de bonne pratique :

- Modales accessibles :

- Ajouter role="dialog", lier le titre avec aria-labelledby.
 - Gérer le focus avec React :

```
useEffect(() => {
  if (isOpen) modalRef.current.focus();
}, [isOpen]);
```

- Redirections accessibles :

Informez clairement les utilisateurs des changements. Par exemple :

```
<a href="new-page.html" aria-label="Aller à la nouvelle page, lien ouvrant dans un nouvel onglet">
  Ouvrir
</a>
```

TP 1

- **Tâches: Sur un formulaire :**

- **Alternatives textuelles :**

- Ajoutez des labels pour chaque champ (<label>), associés correctement à chaque champ via l'attribut `for` et `id`.
 - Ajoutez un texte alternatif pour le bouton de soumission (par exemple, via `aria-label` si nécessaire).

- **Navigation au clavier :**

- Assurez-vous que le formulaire est entièrement navigable au clavier (en utilisant la touche Tab).
 - Mettez en place un focus visible sur les éléments interactifs.

- **Contraste et taille de texte :**

- Vérifiez que le contraste du texte respecte les bonnes pratiques (minimum 4.5:1).
 - Assurez-vous que la taille de texte est relative (utilisez `rem` ou `em`).

- **Messages d'erreur clairs :**

- Implémentez un message d'erreur sous le champ email si l'adresse est invalide.
 - Associez chaque message d'erreur avec l'élément correspondant en utilisant `aria-describedby`.

TP 2 : Créer des composants génériques accessibles

- **Objectif:**

Le but de ce TP est de créer des composants génériques réutilisables respectant les bonnes pratiques d'accessibilité. Vous allez implémenter un bouton accessible et une modale accessible avec un focus management, des rôles ARIA et une gestion des messages d'erreur.

TP 2 : Créer des composants génériques accessibles

- **Contexte 1**

- **Composant Bouton Accessible**

- Créez un composant "AccessibleButton" qui pourra être utilisé partout dans l'application. Ce bouton doit être accessible via le clavier et doit fournir des informations sur son action pour les technologies d'assistance.
 - Implémentez un focus visible pour ce bouton.
 - Utilisez des bonnes pratiques pour le rendu textuel (rôle explicite, actions claires).

TP 2 : Créer des composants génériques accessibles

- **Contexte 2**

Créez un composant AccessibleModal qui doit afficher une modale lorsque l'utilisateur interagit avec un bouton (par exemple, en cliquant sur un bouton pour ouvrir la modale). La modale doit être accessible via le clavier et les technologies d'assistance.

- **Comportements attendus :**

- Lors de l'ouverture de la modale, le focus doit être déplacé à l'intérieur de la modale.
- La modale doit avoir un rôle dialog et un titre associé via aria-labelledby.
- La modale doit pouvoir être fermée via le clavier (par exemple, en appuyant sur la touche Esc).
 - Tâches attendues
 - Implémentez une gestion du focus pour que l'utilisateur reste concentré sur la modale lorsqu'elle est ouverte.
 - Ajoutez une gestion du clavier pour permettre la fermeture de la modale via la touche Esc

Les fonctions pures



- Fonction pure

```
function add(a, b) {  
    return a + b;  
}
```

- **Toujours la même sortie pour la même entrée :** Peu importe combien de fois vous appelez `add(2, 3)`, il retournera toujours 5. La sortie dépend uniquement des valeurs des arguments a et b.
- **Pas d'effets secondaires :** Cette fonction ne modifie aucun état externe. Elle ne dépend que des valeurs de ses arguments et ne provoque aucun effet observable en dehors de sa propre exécution.

- Fonction impure

```
let result = 0;  
  
function impureAdd(a) {  
    result += a;  
    return result;  
}
```

- **Effets secondaires :** Cette fonction modifie un état externe (result). À chaque appel, elle change la valeur de `result`, ce qui rend son comportement dépendant de l'état global, pas seulement de ses arguments.
- **Pas de sortie déterministe :** Bien que la sortie de `impureAdd` dépende de son argument `a`, elle dépend également de l'état de `result`, ce qui peut changer à chaque appel. Ainsi, elle ne garantit pas toujours la même sortie pour la même entrée.

Qu'appelle-t-on 'pure' ?



Dans le contexte 'React', on parle de composants pures ou impures.

- Les composants fonctionnels (donc les fonctions qui retournent du jsx) sont dits pures
- Les composants *class* sont dits impures dans certains cas
- En effet: un composant fonctionnel est généralement considéré comme pure car:
 - **Immutabilité des props** : Dans un composant fonctionnel, les props sont passées comme arguments à la fonction. Comme les props ne sont pas modifiables à l'intérieur de la fonction, cela garantit que le composant fonctionnel n'affectera pas involontairement les données en dehors de son propre contexte.
 - **Aucun état interne implicite** : Les composants fonctionnels n'ont pas d'état interne, ce qui signifie qu'ils ne conservent pas de données entre les rendus. Ils dépendent uniquement des props et de leur propre logique interne pour déterminer leur rendu, ce qui les rend prévisibles et faciles à comprendre.
 - **Pas d'effet secondaire direct**. Grâce au hook *useEffect*, ils peuvent gérer des effets secondaires de manière isolée, assurant ainsi un rendu déterministe et prévisible tout en permettant la gestion des côtés effets de manière séparée.

La sécurité



Pour se protéger des attaques par injection en React (et en JavaScript en général), il est crucial de comprendre les risques potentiels et de mettre en œuvre des pratiques de sécurité appropriées. Voici quelques recommandations pour prévenir les attaques par injection :

- **Utiliser des frameworks de routage sécurisés** : Utilisez des frameworks de routage sécurisés comme React Router pour gérer la navigation dans votre application. Assurez-vous de ne pas intégrer directement des paramètres d'URL non validés dans vos requêtes ou vos rendus JSX.
- **Échapper les données sensibles** : Lorsque vous affichez des données dynamiques dans votre interface utilisateur, assurez-vous d'échapper correctement ces données pour éviter toute exécution de code malveillant. Utilisez des outils comme `dangerouslySetInnerHTML` avec prudence et assurez-vous de ne pas afficher de contenu utilisateur non filtré.
- **Utiliser des bibliothèques d'échappement de données** : Utilisez des bibliothèques d'échappement de données sécurisées telles que `escape-html` pour échapper les caractères spéciaux dans les chaînes de caractères que vous affichez dans votre interface utilisateur.

La sécurité



- **Validation des entrées utilisateur** : Validez toutes les entrées utilisateur **côté client et côté serveur** pour vous assurer qu'elles correspondent aux attentes et qu'elles ne contiennent pas de données malveillantes. Utilisez des outils de validation comme Joi, Yup, ou Express Validator côté serveur, et des méthodes de validation personnalisées ou des bibliothèques de validation côté client.
- **Paramètres de requête sécurisés** : Lorsque vous utilisez des paramètres de requête dans vos requêtes HTTP, assurez-vous de les valider et de les échapper correctement avant de les utiliser dans vos requêtes SQL ou NoSQL pour éviter les attaques d'injection SQL ou NoSQL.
- **Utiliser des requêtes paramétrées** : Si vous interagissez avec une base de données relationnelle, utilisez des requêtes paramétrées ou des ORM qui gèrent automatiquement l'échappement des données pour vous, afin de réduire le risque d'injection SQL.
- **Sensibilisation à la sécurité** : Éduquez-vous et sensibilisez votre équipe aux meilleures pratiques de sécurité en matière de développement web. Tenez-vous informé des dernières menaces et techniques d'attaque, et assurez-vous que votre équipe suit des pratiques de développement sécurisé.

Les hooks



- **useTransition** : Permet de déclarer des transitions pour les mises à jour d'état asynchrones.
- **useDeferredValue** : Retarde la mise à jour de la valeur d'un hook pour améliorer les performances.
- **useMutableSource** : Permet d'accéder à une source de données mutable à partir d'un contexte de données, notamment utile pour les bibliothèques tierces.
- **useInterstate** : Alternative à useState pour le partage d'état entre plusieurs composants d'un arbre sans prop drilling.
- **useEventSource** : Hook pour gérer les connexions avec des sources d'événements côté client.
- **useIdle** : Hook pour suivre l'état d'inactivité de l'utilisateur sur une page web.
- **useLayoutAnimation** : Permet de contrôler les animations de disposition dans React Native.