

**Creation:**

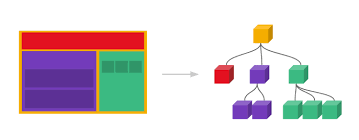
npx create-react-app nom-app

<https://create-react-app.dev/docs/getting-started>

npm create vite@latest

<https://vitejs.dev/guide/>

**Interface :** JSX, une syntaxe qui ressemble au HTML, mais avec des petites différences : ***className*** remplace ***class***, toute les balises doivent être fermées (img, input, …), les variables sont insérées entre accolades ***{maVariable}****.*

**Composants :**

L’interface utilisateur est un composant, qui se divise en composants fils et ainsi de suite.

Le composant parent envoi les données aux composant fils par les **props.** On peut créer props en ajoutant des attributs lorsqu’on implémente le composant fils.

**Exemple:**

<Post key={post.id} idPost={post.id} idAuthor={post.idAuthor} updatePost={openModal} supprimerCePost={supprimerUnPost} texte={post.contenu} lienImage={post.imgUrl} horodatage={post.horodatage} />

Dans l’exemple les props contients : idPost, idAuthor, updatePost, supprimerCePost, texte, lienImage, horodatage.

<Commentaire key={commentaire.id} userId={user.id} idAuthor={commentaire.idAuthor} textCom={commentaire.message} idComm={commentaire.id} supprimerCeComm={supprimerUnComm} />

Dans cet exemple les props contients : userId, idAuthor, textCom, idComm, supprimerCeCom.

Pour les consommer dans le composant fils :

1e façon : {props.idPost}, {props.idAuthor},….

export default function Commentaire(props) {

/\*

Le reste du code ici…

\*/

  return (

    <div>

      {props.textCom}

      {(props.userId===props.idAuthor)&&

      <button onClick={supprimerComm} className="btn btn-delete"><i className="bi bi-trash3"></i></button>}

    </div>

  )

}

2e façon : pour éviter de répéter le mot props

const { idPost, texte, lienImage } = props;

Ensuite les insérer dans JSX sans le mot props :

{idPost}, {idAuthor},…

export default function Commentaire(props) {

const { textCom, userId, idAuthour } = props;

/\*

Le reste du code ici…

\*/

  return (

    <div>

      {textCom}

      {(userId===idAuthor)&&

      <button onClick={supprimerComm} className="btn btn-delete"><i className="bi bi-trash3"></i></button>}

    </div>

  )

}

Ou comme ça:

export default function Commentaire({ textCom, userId, idAuthour }) {

/\*

Le reste du code ici…

\*/

  return (

    <div>

      {textCom}

      {(userId===idAuthor)&&

      <button onClick={supprimerComm} className="btn btn-delete"><i className="bi bi-trash3"></i></button>}

    </div>

  )

}

**State :** tout ce qui doit être changé, doit être mis dans un state : le contenu d’un input, la liste des composants, …

    const [listPost, setListPost] = useState([])

    const [inputContenu, setInputContenu] = useState('');

le changement de la liste des postes ***listPost*** doit passer par la fonction ***setListPost***, la valeur initiale du state est ***[ ]*** . Le changement du contenu de l’input ***inputContenu*** doit passer la fonction ***setInputContenu,*** le state initial est ***‘’***.

**useEffect :** Pour lancer un fonction au lancement d’un composant, comme pour envoyer une requête http ***fetch***, on utilise useEffect, avec un tableau de dépendances vide, exemple :

useEffect(() => {

        fetch('http://localhost:3001/api/posts', {

            headers: {

                Authorization: `Bearer ${token}`,

            },

        })

            .then(res => res.json())

            .then(data => setListPost(data))

            .catch(err => alert(err))

    }, [])

**State Management :** Quand il y a des données utilisées un peu partout dans une application, on implémente le state management, les deux solutions les plus populaires : **context api** pour les applications qui n’atteint pas un certain volume, et **redux** pour les grandes applications.

L’implémentation de chacune des solutions passe par une structure spécifique.