TP3 : Les concepts avancés de React

Objectif

Créer une application, nommée **"ReactBook"**, simulant un réseau social avec gestion des posts, authentification (simulée), navigation, et bascule de thème en appliquant des concepts avancés de React, tels que : hooks, événements, état complexe, contexte global et routage, etc.

Partie 1 : Introduction aux Hooks et à l'État de Base

Arborescence à mettre en place

```
src/
 components/
  PostCard.jsx
  Feed.jsx
  Header.jsx
  UserProfile.jsx
 App.jsx
 main.jsx
```

État Local (`useState`) et Événements

1. Dans le fichier `PostCard.jsx`, créer la structure de base d'un composant `PostCard` qui accepte des **props** (`author`, `content`, `initialLikes`).
2. Dans `PostCard`, utiliser `useState` pour gérer le nombre de J'aime (initialisé par `initialLikes`).
3. Ajoutez un bouton "J'aime ♡" qui, au clic, incrémente le nombre de likes dans l'état local du `PostCard`.
4. Pourquoi est-il essentiel, lors de la mise à jour des likes, d'utiliser la forme fonctionnelle `setLikes(prevLikes => ...)` ?
5. Pourquoi est-ce particulièrement important si un utilisateur **double-clique très rapidement** sur le bouton "J'aime" ?
6. Ajouter un bouton "Réinitialiser" qui remet le nombre de likes à la valeur initiale fournie par la **prop** `initialLikes`.
7. Afficher le composant `PostCard`, importé depuis `src/components/PostCard.jsx`, avec des valeurs d'exemple dans `App.jsx`.

8. Que se passe-t-il si vous modifiez directement la variable d'état sans utiliser la fonction `setX()` ? Expliquez ce qui se passe au niveau de l'affichage et au niveau de la variable elle-même.

9. Pourquoi le composant se re-render-t-il après un `setState()` ?

Communication Parent/Enfant et `useEffect`

10. Dans un composant parent `Feed`, créez un bouton "Ajouter un Post" qui, au clic, affiche simplement en console "Ouverture du formulaire".

    [Fichier : `src/components/Feed.jsx`]

11. Passer cette fonction de callback au `Header` de votre application via les **props**.

    Indication : Définir dans `src/components/Feed.jsx`, passer à `src/components/Header.jsx`, utiliser dans `Header.jsx`.

12. Dans le composant `Feed`, utilisez `useState` pour suivre le nombre total de posts affichés.

13. Ajoutez un `useEffect` qui met à jour le titre du navigateur (`document.title`) pour afficher : `"ReactBook | [X] Posts"` (où X est le nombre actuel de posts).

14. Définissez le tableau de dépendances de ce `useEffect`. Que se passerait-il si vous laissiez ce tableau vide (`[]`) ou si vous l'omettiez complètement ?

15. Quand s'exécute le code dans `useEffect()` ? (Citez les trois cas).

16. À quoi sert le `return` dans un `useEffect` ? (Expliquez son rôle de nettoyage).

Requêtes API et Cycle de Vie

17. Dans un fichier `UserProfile.jsx`, créez un composant `UserProfile` qui récupère les infos d'un utilisateur sur `https://jsonplaceholder.typicode.com/users/1`.

18. Affichez `name`, `email` et `company.name` de l'utilisateur récupéré.

19. Affichez un message ‘‘`Chargement...`’’ pendant la requête (utilisez un `useState` pour la variable `isLoading`).

20. Que se passerait-il si on oublie le tableau de dépendances dans le `useEffect` contenant la requête ?

21. Expliquez comment vous utiliseriez la fonction de nettoyage pour annuler l'appel API si l'utilisateur changeait de page (démontage du composant) avant que la réponse ne soit reçue.

Partie 2 : Configuration Git et GitHub (Travail Monôme)

Cette partie vise à simuler et à documenter le flux de travail d'une équipe, même si vous travaillez seul, afin de respecter les standards de l'industrie.

Initialisation

1. Initialisez un dépôt Git local dans le dossier de votre projet (déjà existant "partie 1").
2. Créez un fichier `.gitignore` pour exclure les dossiers non nécessaires (`node_modules`, `dist`, etc.).
3. Effectuez le commit initial.
4. Créez un nouveau dépôt sur GitHub et liez-le à votre dépôt local. Poussez le code initial sur la branche `main` (ou `master`).

Flux de Travail (Git Flow Simplifié)

Pour chaque partie majeure du TP, vous devrez suivre les étapes suivantes :

5. Créez une nouvelle **branche de fonctionnalité** pour la Partie 3.
6. Effectuez vos développements sur cette branche avec des commits réguliers et clairs.
7. Une fois la partie terminée, poussez la branche vers GitHub.
8. Simulez une **Pull Request (PR)** sur GitHub. Documentez dans la description de la PR ce que vous avez fait et comment le tester.
9. Fusionnez la PR dans la branche `main` puis supprimer cette branche.
10. Quels sont les avantages de créer des branches de fonctionnalités (feature branches) avant de commencer le développement, même dans un contexte de travail individuel ?

Partie 3 : Gestion des Événements et Communication

Événements de Saisie et Événements Synthétiques

1. Dans un fichier `InputLogger.jsx`, Créez un composant avec un `<input type="text" />`.
2. Affichez la valeur tapée dans un `<p>` via un handler `onChange` utilisant l'état local.

3. Dans le handler `onChange`, ajoutez `console.log(event)` pour observer la structure du **SyntheticEvent** de React.
4. Dans `App.jsx`, importez et rendez `<InputLogger />`.
5. Quelle est la différence fondamentale entre un événement DOM classique (que l'on obtiendrait avec `document.addEventListener`) et un `SyntheticEvent` ?
6. Pourquoi React gère-t-il ses propres événements synthétiques au lieu d'utiliser directement ceux du navigateur ?

Formulaires Contrôlés et Prévention

7. Dans un fichier `LoginForm.jsx`, Créez un composant avec deux champs contrôlés par `useState` : `email` et `password`.
8. Créez un handler `onSubmit` pour le formulaire. Il doit utiliser `event.preventDefault()` puis afficher en console l'objet `{ email, password }`.
9. Après soumission réussie (affichée en console), mettez à jour un état local pour afficher un message ‘‘`Connexion réussie !`’’ dans un `<p>` pendant 3 secondes.
10. Dans `App.jsx`, importez et rendez `<LoginForm />`.
11. Pourquoi faut-il absolument empêcher le comportement par défaut du formulaire (`event.preventDefault()`) dans une application Single Page Application (SPA) comme ReactBook ?

Communication Parent → Enfant → Parent (Remontée d'État)

12. **Créer le formulaire enfant** `MessageForm.jsx` **:** Ce composant doit contenir un champ texte et un bouton “Publier”. Il ne doit **pas** stocker la liste des messages, mais exposer un callback nommé `onAddMessage` via ses **props**.
13. **Créer le parent** `MessageBoard.jsx` **:** Ce composant doit stocker un tableau de messages dans son état local. Il doit afficher cette liste.
14. **Passage de callback :**
    - Définir une fonction `addMessage(newMessage)` dans `src/components/MessageBoard.jsx` qui met à jour la liste des messages.
    - Transmettre cette fonction à l'enfant : `<MessageForm onAddMessage={addMessage} />`
    - Dans l'enfant (`MessageForm.jsx`), appeler `props.onAddMessage(message)` à la soumission (après `preventDefault()`).
15. Dans `App.jsx`, importez et rendez `<MessageBoard />`.

16. Pourquoi les **props** ne peuvent-elles pas être modifiées directement par le composant enfant (`MessageForm`) ?

17. Comment React sait-il qu'il doit re-render le parent (`MessageBoard`) quand on ajoute un message via le callback ? (Expliquez le rôle central de `setState` dans cette remontée d'état).

Conseil pratique : Pour chaque composant, ajoutez temporairement des `console.log("render <NomComposant>")` pour visualiser quand et pourquoi il se re-render (utile pour comprendre la boucle : `événement` → `setState` → `render`).

Partie 3 : Contexte global et état partagé

Gestion d'État Structuré et Avancé

1. **Définition du Reducer :** Créez un fichier `feedReducer.js`, où vous allez définir la fonction `feedReducer(state, action)` qui gère un état complexe (un tableau d'objets post).

2. Implémentez les actions (type et payload) suivantes :

   - `'ADD_POST'` : Ajoute un nouveau post au tableau (créé en respectant l'immutabilité).
   - `'TOGGLE_LIKE'` : Inverse le statut "liked" pour un post donné (recherche par ID).
   - `'DELETE_POST'` : Supprime un post par son ID.

3. Dans le composant `Feed.jsx`, remplacez l'état local par `useReducer` avec votre `feedReducer`.

4. Affichez une liste interactive des posts (les boutons J'aime/Supprimer doivent utiliser la fonction `dispatch`).

5. Pourquoi `useReducer` est-il préférable à `useState` pour gérer ces trois actions simultanément ?

6. Que représente la fonction "reducer" dans ce contexte ? Expliquez sa nature et son rôle dans la transition d'état.

Hooks Personnalisés (`Custom Hooks`) et Logique de Formulaire

1. **Création du Hook** `useFormInput` **:** Créez une fonction `useFormInput(initialValue)` qui utilise `useState` en interne pour stocker une valeur de champ, et retourne un objet contenant la `value` et une fonction `onChange` pré-configurée pour cet `<input />`.

2. **Formulaire de Création (`CreatePostForm.jsx`) :**
   - Créez un composant `CreatePostForm`.
   - Utilisez votre `useFormInput` pour gérer l'état du champ de texte du nouveau post.
   - Lors de la soumission (`onSubmit`), utilisez la fonction `dispatch` du `feedReducer` (passée en prop depuis `Feed.jsx`) pour déclencher l'action `'ADD_POST'`.
3. En quoi `useFormInput` factorise-t-il la **logique** et non l'état ? Si deux composants utilisent ce Hook, partagent-ils la même variable d'état ? Expliquez.
4. Quelle est la différence entre un Hook (comme `useFormInput`) et une simple fonction JS utilitaire ?
5. Pourquoi un Hook doit-il toujours commencer par `use` ?

Gestion de l'état global (Context API)

1. Dans un fichier `src/context/AuthContext.jsx`, créez un `AuthContext` et un `AuthProvider`.
2. Dans `AuthProvider`, gérez l'état d'authentification avec `useState`. L'état `user` doit être initialisé à `null` et le Provider doit fournir l'objet de contexte via sa `value` avec les éléments suivants :
   - `user` : l'utilisateur actuellement connecté (objet ou `null`).
   - `login(userObject)` : une fonction qui simule la connexion en mettant à jour l'état `user`.
   - `logout()` : une fonction qui remet `user` à `null`.
3. Dans le composant `Header.jsx`, consommez le contexte avec `useContext(AuthContext)`.
4. Affichez dynamiquement dans le `Header` :
   - ``Connecté en tant que [nom d'utilisateur]'' si `user` existe.
   - ``Invité'' si `user` vaut `null`.
5. Créez un `ThemeContext` et un `ThemeProvider` dans un fichier `src/context/ThemeContext.jsx`.
6. Dans `ThemeProvider`, utilisez `useReducer` pour gérer l'état du thème. L'état initial doit être `{theme: 'light'}`.
7. Le reducer doit gérer une seule action : `'TOGGLE_THEME'`, qui inverse la valeur du thème (`'light'` ou `'dark'`).
8. Le Provider doit fournir via sa `value` :

- la valeur actuelle de `theme` (chaîne de caractères).
- la fonction `dispatch` pour permettre de changer le thème depuis n'importe quel composant.

9. Pourquoi est-il nécessaire d'imbriquer les Providers (`AuthProvider` et `ThemeProvider`) dans `App.jsx` ?
10. Si l'on souhaite accéder à la fois à l'utilisateur et au thème dans un même composant, pourquoi est-il préférable d'avoir **deux Contextes distincts** plutôt qu'un seul contexte géant ?
11. Que se passe-t-il si plusieurs Providers **identiques** sont imbriqués (par exemple, deux `ThemeProvider`) ? Quelle valeur le composant le plus profond consommera-t-il ?

Partie 5 : Routage et Protection des Routes

1. Enveloppez l'application dans `BrowserRouter` (dans `main.jsx` ou `App.jsx`) et configurez le composant `Routes`.
2. Définissez les chemins suivants (créez les composants de page `FeedPage.jsx`, `LoginPage.jsx`, `ProfilePage.jsx` dans `src/pages/`) :
   - `/ (Accueil)` : Rend le composant `FeedPage` (où réside la liste des posts).
   - `/login` : Rend le composant `LoginPage` (où se trouve le formulaire de connexion).
   - `/user/:username` : Rend le composant `ProfilePage` (Profil d'un utilisateur, paramétré).
3. Dans `LoginPage`, simulez une connexion réussie (mise à jour de l'état dans `AuthContext`) et utilisez le Hook `useNavigate()` pour rediriger l'utilisateur vers la route `/`.
4. Dans `PostCard.jsx`, l'auteur doit être cliquable. Créez un lien (`<Link>`) vers la route `/user/[nom de l'auteur]`.
5. Dans `ProfilePage.jsx`, utilisez le Hook `useParams()` pour extraire le `username` de l'URL et affichez-le dans un titre de section.
6. Si vous vouliez ajouter un filtre de recherche (`/feed?search=react`) plutôt que d'extraire l'ID dans l'URL, quel Hook de `react-router-dom` utiliseriez-vous à la place de `useParams()` ?
7. Créez un composant `ProtectedRoute` qui accepte `children` en prop.
8. Ce composant doit lire l'état de `AuthContext`. S'il n'y a pas d'utilisateur connecté, il doit utiliser le composant `<Navigate to="/login" replace />` pour bloquer l'accès.

9. Rendez la route `/ (FeedPage)` et la route `/user/:username` accessibles uniquement aux utilisateurs connectés en les enveloppant avec `<ProtectedRoute>`.

