Étape 6 : Context API et Hooks personnalisés

Objectif

Finaliser l'application avec un état global (utilisateur actuel) et des fonctionnalités transversales (thème, persistance).

Concepts pratiqués

- Context API pour l'état global
- useContext pour consommer le contexte
- Hooks personnalisés (custom hooks)
- Persistance avec localStorage

Approche simplifiée : Sélecteur d'utilisateur

> **Pourquoi pas de login/mot de passe ?**
>
> Un vrai système d'authentification (login, register, JWT, etc.) est complexe et hors scope pour cette formation. À la place, on utilise un **sélecteur d'utilisateur** qui permet de :
>
> - Simuler différents utilisateurs
> - Tester les rôles (admin, employé)
> - Apprendre les mêmes concepts (Context, état global)
>
> C'est une approche réaliste pour un prototype ou une démo.

À créer

1. `UserContext.jsx` - Contexte utilisateur

Un contexte global pour :

- Stocker l'utilisateur actuellement sélectionné
- Fournir une fonction pour changer d'utilisateur
- Indiquer si l'utilisateur est admin
- Accessible partout dans l'application

2. `UserSelector.jsx` - Sélecteur d'utilisateur

Un composant (dropdown) qui :

- Affiche la liste des employés
- Permet de sélectionner "qui je suis"
- Met à jour le contexte

3. `useLocalStorage.js` - Hook personnalisé

Un hook qui :

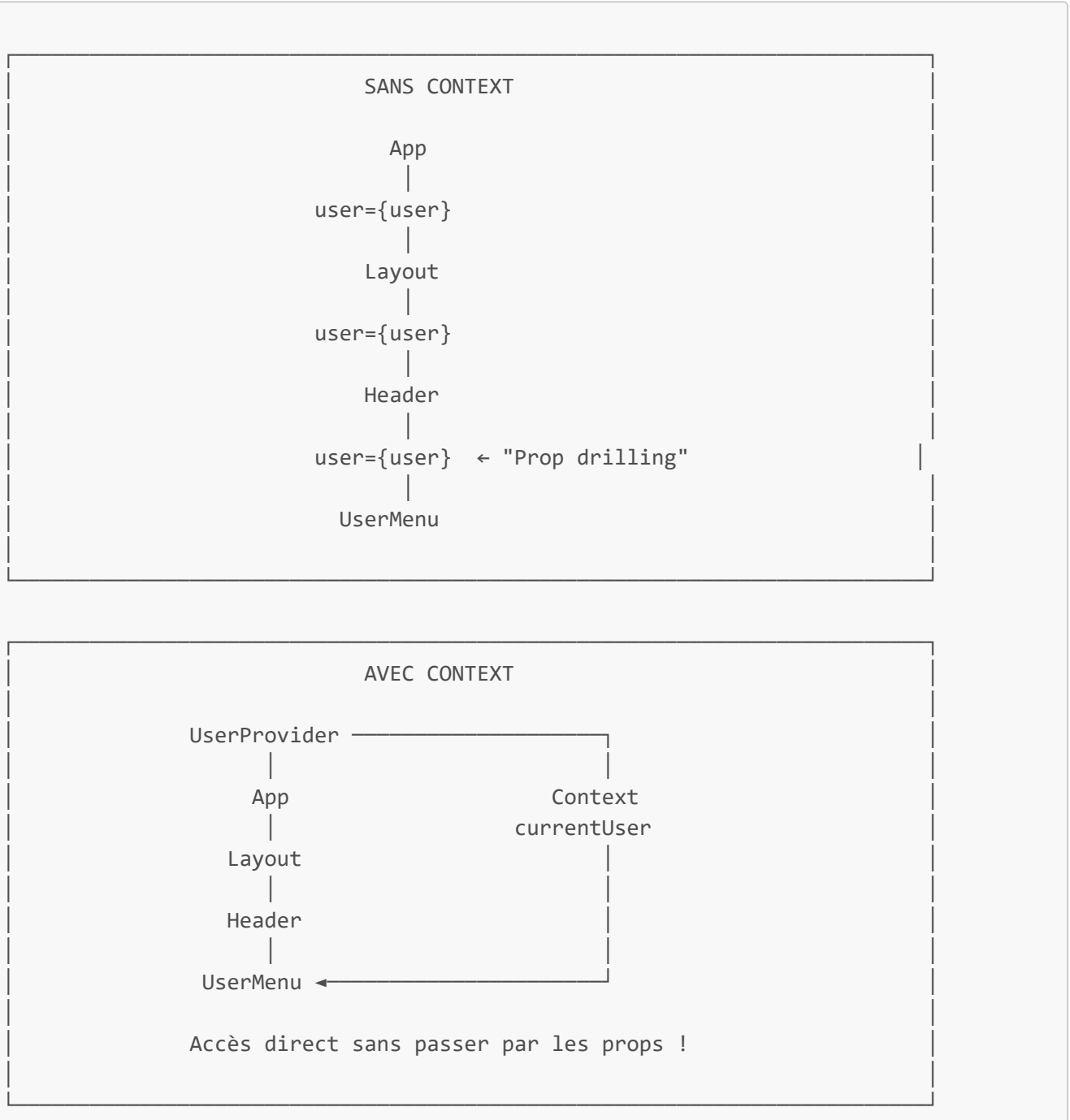
- Synchronise un state avec localStorage
- Persiste les données entre les sessions
- Gère la sérialisation/désérialisation

4. Thème clair/sombre

Une fonctionnalité qui :

- Toggle entre thème clair et sombre
- Persiste le choix via useLocalStorage
- Applique les styles CSS correspondants

Context API : Quand l'utiliser ?



Données employés avec rôles

Modifiez `employees.js` pour ajouter un champ `isAdmin` :

```
export const employees = [
  {
    id: 1,
    firstName: "Victor",
    lastName: "Besson",
    // ... autres champs
    isAdmin: true,  // Peut poster des annonces
  },
  {
    id: 2,
    firstName: "Alice",
    lastName: "Martin",
    // ... autres champs
    isAdmin: false,  // Employé standard
  },
  // ...
];
```

Indices

▶ 💡 Structure du UserContext

```
import { createContext, useContext, useState } from 'react';
import { employees } from '../data/employees';

// 1. Créer le contexte
const UserContext = createContext(null);

// 2. Créer le Provider
function UserProvider({ children }) {
  // Par défaut, premier utilisateur ou null
  const [currentUser, setCurrentUser] = useState(employees[0]);

  // Fonction pour changer d'utilisateur
  const selectUser = (userId) => {
    const user = employees.find(e => e.id === userId);
    if (user) {
      setCurrentUser(user);
    }
  };

  // Valeur fournie à tous les descendants
```

```
  const value = {
    currentUser,
    selectUser,
    isAdmin: currentUser?.isAdmin || false,
  };

  return (
    <UserContext.Provider value={value}>
      {children}
    </UserContext.Provider>
  );
}

// 3. Hook personnalisé pour consommer
function useUser() {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error('useUser doit être utilisé dans un UserProvider');
  }
  return context;
}

export { UserProvider, useUser };
```

▶ 💡 Composant UserSelector

```
import { useUser } from '../contexts/UserContext';
import { employees } from '../data/employees';

function UserSelector() {
  const { currentUser, selectUser } = useUser();

  return (
    <div className="user-selector">
      <label htmlFor="user-select">Connecté en tant que :</label>
      <select
        id="user-select"
        value={currentUser?.id || ''}
        onChange={(e) => selectUser(Number(e.target.value))}
      >
        {employees.map(employee => (
          <option key={employee.id} value={employee.id}>
            {employee.firstName} {employee.lastName}
            {employee.isAdmin && ' (Admin)'}
          </option>
        ))}
      </select>
    </div>
  );
}
```

▶ Hook useLocalStorage

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

function useLocalStorage(key, initialValue) {
  // État initialisé depuis localStorage
  const [value, setValue] = useState(() => {
    try {
      const stored = localStorage.getItem(key);
      return stored ? JSON.parse(stored) : initialValue;
    } catch {
      return initialValue;
    }
  });

  // Synchroniser avec localStorage à chaque changement
  useEffect(() => {
    try {
      localStorage.setItem(key, JSON.stringify(value));
    } catch (error) {
      console.error('Erreur localStorage:', error);
    }
  }, [key, value]);

  return [value, setValue];
}

export default useLocalStorage;
```

▶ Utiliser le contexte pour conditionner l'affichage

```
// Dans AnnouncementList.jsx
import { useUser } from '../contexts/UserContext';

function AnnouncementList() {
  const { isAdmin } = useUser();

  return (
    <div>
      {/* Le formulaire n'apparaît que pour les admins */}
      {isAdmin && <AnnouncementForm onAdd={handleAddAnnouncement} />}

      {/* Liste des annonces */}
      {announcements.map(a => <AnnouncementCard key={a.id} {...a} />)}
    </div>
  );
}
```

▶ ThemeContext avec persistance

```
import { createContext, useContext } from 'react';
import useLocalStorage from '../hooks/useLocalStorage';

const ThemeContext = createContext(null);

function ThemeProvider({ children }) {
  // Le thème est persisté dans localStorage
  const [theme, setTheme] = useLocalStorage('theme', 'light');

  const toggleTheme = () => {
    setTheme(prev => prev === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      <div className={`app theme-${theme}`}>
        {children}
      </div>
    </ThemeContext.Provider>
  );
}

function useTheme() {
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error('useTheme doit être utilisé dans un ThemeProvider');
  }
  return context;
}

export { ThemeProvider, useTheme };
```

Points d'attention

⚠ Provider doit envelopper l'application

```
// Dans main.jsx ou App.jsx
<UserProvider>
  <ThemeProvider>
    <App />
  </ThemeProvider>
</UserProvider>
```

⚠ Vérifier que le context existe

```
function useUser() {
  const context = useContext(UserContext);
```

```
  // Si useUser() est appelé en dehors du Provider
  if (!context) {
    throw new Error('useUser doit être utilisé dans UserProvider');
  }

  return context;
}
```

⚠ localStorage et SSR

```
// localStorage n'existe pas côté serveur (Next.js, etc.)
const [value, setValue] = useState(() => {
  // Vérifier qu'on est côté client
  if (typeof window === 'undefined') {
    return initialValue;
  }
  const stored = localStorage.getItem(key);
  return stored ? JSON.parse(stored) : initialValue;
});
```

> Note : Avec Vite (notre cas), ce n'est pas un problème car on est toujours côté client.

Mise à jour des composants existants

Header.jsx

Remplacez l'utilisateur "en dur" par le contexte :

```
// Avant (étape 1)
const currentUser = { name: "Victor", role: "Développeur" };

// Après (étape 6)
import { useUser } from '../contexts/UserContext';

function Header() {
  const { currentUser } = useUser();
  // ...
}
```

AnnouncementForm.jsx

Utilisez l'utilisateur du contexte comme auteur :

```
import { useUser } from '../contexts/UserContext';
```

```
function AnnouncementForm({ onAdd }) {
  const { currentUser } = useUser();

  const handleSubmit = (e) => {
    e.preventDefault();
    onAdd({
      title,
      content,
      category,
      author: `${currentUser.firstName} ${currentUser.lastName}`,
      authorRole: currentUser.role,
    });
  };
  // ...
}
```

Critères de validation

- [ ] Le sélecteur d'utilisateur fonctionne
- [ ] L'utilisateur actuel est accessible partout via useUser()
- [ ] Le formulaire d'annonce n'apparaît que pour les admins
- [ ] Le thème toggle fonctionne
- [ ] Le choix du thème est persisté (recharger la page)
- [ ] L'utilisateur sélectionné est persisté (bonus)
- [ ] Pas d'erreur si context non disponible

Pour aller plus loin

- Persister l'utilisateur sélectionné dans localStorage
- Ajouter un avatar dans le sélecteur
- Créer un hook `useAnnouncements` pour centraliser la logique
- Ajouter des animations au changement de thème