React JS 1 Master SWM ISITCOM

Séance 5 : Navigation Multi-Pages et Gestion Avancée des Données

Dr Abdelweheb GUEDDES

15 novembre 2025

Table des matières

1 Introduction : Vers des Applications Réelles

Jusqu'à présent, nous avons construit une application Rick & Morty qui fonctionne sur une seule page. Dans le monde réel, les applications modernes sont organisées en **plusieurs pages** (ou vues) avec une navigation fluide, et communiquent intensivement avec des **APIs externes**.

Cette séance aborde deux piliers essentiels des applications React professionnelles :

1. **React Router** : Pour créer une application multi-pages avec navigation côté client
2. **RTK Query** : Pour gérer les appels API de manière optimale avec cache, invalidation automatique, et gestion des états de chargement

Objectifs de la Séance 5

1. Maîtriser le routing avec React Router (navigation, paramètres d'URL, routes imbriquées)
2. Comprendre RTK Query et son intégration avec Redux Toolkit
3. Gérer le cache, la synchronisation et l'invalidation des données
4. Construire une architecture scalable pour les appels API

2 Partie 1 : Théorie (1h30)

2.1 React Router : Navigation Côté Client

2.1.1 Le Problème : Applications Single-Page

Par défaut, React crée des **Single Page Applications (SPA)** : toute l'application vit dans une seule page HTML. Quand l'utilisateur clique sur un lien, JavaScript change le contenu sans recharger la page.

Les avantages :

— Navigation instantanée (pas de rechargement)

— Meilleure expérience utilisateur

— Moins de charge serveur

Le défi :

— Comment gérer différentes "pages" ?

— Comment synchroniser l'URL avec le contenu affiché ?

— Comment permettre le partage d'URL et les boutons précédent/suivant du navigateur ?

2.1.2 La Solution : React Router

React Router est la bibliothèque standard pour gérer la navigation dans React. Elle synchronise l'URL du navigateur avec les composants à afficher.

Les Concepts Fondamentaux

`<BrowserRouter>` Le composant racine qui active le routing. Il doit envelopper toute votre application.

`<Routes>` Conteneur pour définir vos routes. Il examine toutes les routes enfants et affiche celle qui correspond à l'URL actuelle.

`<Route>` Définit une correspondance entre un chemin d'URL et un composant à afficher.

`<Link>` Remplace la balise `<a>` pour la navigation sans rechargement de page.

`<NavLink>` Comme `<Link>`, mais peut avoir un style différent quand il est actif.

`useParams()` Hook pour récupérer les paramètres dynamiques de l'URL (ex : `/character/:id`).

`useNavigate()` Hook pour naviguer programmatiquement (dans du code JavaScript).

2.1.3 Exemple de Base

```jsx
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Accueil</Link>
        <Link to="/about">À propos</Link>
      </nav>

      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage />} />
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Listing 1 – Structure de routing simple

2.1.4 Routes avec Paramètres Dynamiques

Pour créer des pages de détails (ex : profil d'un utilisateur, détails d'un produit) :

```jsx
import { useParams } from 'react-router-dom';

// Definition de la route
<Route path="/character/:id" element={<CharacterDetail />} />

// Composant qui utilise le parametre
function CharacterDetail() {
  const { id } = useParams(); // Recupere l'ID depuis l'URL

  return <h1>Détails du personnage #{id}</h1>;
}

// Navigation vers cette route
<Link to="/character/42">Voir personnage 42</Link>
```

Listing 2 – Routes dynamiques avec paramètres

2.1.5 Navigation Programmatique

Parfois, vous devez naviguer dans votre code (après une soumission de formulaire, une connexion, etc.) :

```jsx
import { useNavigate } from 'react-router-dom';

function LoginForm() {
  const navigate = useNavigate();

  const handleLogin = async (credentials) => {
    const success = await loginUser(credentials);

    if (success) {
      navigate('/dashboard'); // Rediriger vers le tableau de bord
    }
  };

  return <form onSubmit={handleLogin}>...</form>;
}
```

Listing 3 – Navigation avec useNavigate

2.2 RTK Query : La Révolution des Appels API

2.2.1 Le Problème : La Complexité des Appels API

Gérer des appels API dans React nécessite traditionnellement beaucoup de code répétitif :

```
function CharactersList() {
  const [characters, setCharacters] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  useEffect(() => {
    setLoading(true);
    fetch('https://rickandmortyapi.com/api/character')
      .then(res => res.json())
      .then(data => {
        setCharacters(data.results);
        setLoading(false);
      })
      .catch(err => {
        setError(err.message);
        setLoading(false);
      });
  }, []);

  // ... gestion de l'affichage
}
```

Listing 4 – Approche manuelle avec useState et useEffect

Les problèmes de cette approche :

- Beaucoup de code répétitif (boilerplate)
- Pas de cache : chaque montage du composant refait l'appel
- Pas de synchronisation : si deux composants affichent les mêmes données, deux appels sont faits
- Gestion manuelle du loading et des erreurs
- Pas d'invalidation automatique après une mutation

2.2.2 La Solution : RTK Query

RTK Query est un outil puissant inclus dans Redux Toolkit qui résout tous ces problèmes. C'est comme avoir un "gestionnaire intelligent d'API" qui :

- **Cache automatiquement** les résultats
- **Partage les données** entre tous les composants
- **Gère automatiquement** les états de chargement et d'erreur
- **Rafraîchit automatiquement** les données quand nécessaire
- **Déduplique** les requêtes identiques

2.2.3 Les Concepts Clés de RTK Query

L'API Service C'est un objet qui définit tous vos endpoints (points d'accès à votre API) :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';

export const rickAndMortyApi = createApi({
  reducerPath: 'rickAndMortyApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://rickandmortyapi.com/api'
  }),
  endpoints: (builder) => ({
    // Endpoint pour recuperer tous les personnages
    getCharacters: builder.query({
```

```
      query: () => '/character',
    }),

    // Endpoint pour recuperer un personnage par ID
    getCharacterById: builder.query({
      query: (id) => `/character/${id}`,
    }),
  }),
});

// RTK Query genere automatiquement des hooks
export const {
  useGetCharactersQuery,
  useGetCharacterByIdQuery
} = rickAndMortyApi;
```

Listing 5 – Création d'un API service

Les Hooks Générés Automatiquement Pour chaque endpoint, RTK Query génère un hook qui retourne :

`data` Les données récupérées (undefined au départ)

`isLoading` true pendant le premier chargement

`isFetching` true pendant n'importe quel chargement (y compris rafraîchissement)

`isSuccess` true si la requête a réussi

`isError` true en cas d'erreur

`error` L'objet d'erreur si applicable

`refetch` Fonction pour relancer manuellement la requête

```
function CharactersList() {
  const {
    data,
    isLoading,
    isError,
    error
  } = useGetCharactersQuery();

  if (isLoading) return <p>Chargement...</p>;
  if (isError) return <p>Erreur : {error.message}</p>;

  return (
    <div>
      {data.results.map(char => (
        <div key={char.id}>{char.name}</div>
      ))}
    </div>
  );
}
```

Listing 6 – Utilisation simple d'un hook RTK Query

Le Cache Automatique RTK Query cache automatiquement les résultats. Si un deuxième composant demande les mêmes données :

- Aucun nouvel appel réseau n'est fait
- Les données sont immédiatement disponibles depuis le cache
- Si les données sont "périmées", un rafraîchissement en arrière-plan peut être déclenché

Les Tags pour l'Invalidation Les "tags" sont un système qui permet d'invalider automatiquement le cache :

```
export const tasksApi = createApi({
  reducerPath: 'tasksApi',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Task'], // Declarer les types de tags
  endpoints: (builder) => ({
    getTasks: builder.query({
      query: () => '/tasks',
      providesTags: ['Task'], // Cette requete fournit des "Task"
    }),

    addTask: builder.mutation({
      query: (task) => ({
        url: '/tasks',
        method: 'POST',
        body: task,
      }),
      invalidatesTags: ['Task'], // Cette mutation invalide les "Task"
      // Resultat : apres ajout, getTasks() sera automatiquement re-execute
    }),
  }),
});
```

Listing 7 – Utilisation des tags pour invalidation

2.2.4 Queries vs Mutations

RTK Query distingue deux types d'opérations :

`query` Pour **lire** des données (GET). Mise en cache, partagées, rafraîchissement automatique.

`mutation` Pour **modifier** des données (POST, PUT, DELETE). Ne sont pas mises en cache, peuvent invalider d'autres queries.

```
export const tasksApi = createApi({
  // ... configuration de base
  endpoints: (builder) => ({
    addTask: builder.mutation({
      query: (newTask) => ({
        url: '/tasks',
        method: 'POST',
        body: newTask,
      }),
    }),
  }),
});

export const { useAddTaskMutation } = tasksApi;

// Utilisation dans un composant
function AddTaskForm() {
  const [addTask, { isLoading }] = useAddTaskMutation();

  const handleSubmit = async (taskData) => {
    try {
      await addTask(taskData).unwrap();
      alert('Tâche ajoutée !');
    } catch (error) {
      alert('Erreur : ' + error.message);
    }
  };

  return <form onSubmit={handleSubmit}>...</form>;
```

```
}
```

Listing 8 – Exemple de mutation

2.3 Comparaison : Avant/Après RTK Query

Approche Manuelle	Avec RTK Query
30-40 lignes de code par endpoint	5-10 lignes par endpoint
Gestion manuelle de loading/error	Automatique
Pas de cache (requêtes dupliquées)	Cache automatique et partagé
Invalidation manuelle complexe	Tags d'invalidation automatique
Code dispersé dans les composants	Centralisé dans l'API service
Difficile à tester	Facile à tester et à mocker

TABLE 1 – Comparaison des approches de gestion d'API

3 Partie 2 : Travaux Pratiques (TP) (2h)

Nous allons transformer notre application Rick & Morty en une application multi-pages avec :

- Une page d'accueil avec la liste des personnages
- Une page de détails pour chaque personnage
- Une page des favoris
- Une navigation fluide avec React Router
- Gestion optimale des appels API avec RTK Query

3.1 Étape 1 : Installation des Dépendances

```
npm install react-router-dom
```

Note : RTK Query est déjà inclus dans `@reduxjs/toolkit`, pas besoin d'installation supplémentaire.

3.2 Étape 2 : Créer l'API Service avec RTK Query

Créez le fichier `src/services/rickAndMortyApi.js` :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';

export const rickAndMortyApi = createApi({
  reducerPath: 'rickAndMortyApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://rickandmortyapi.com/api'
  }),
  tagTypes: ['Character'],
  endpoints: (builder) => ({
    // Recuperer tous les personnages avec pagination optionnelle
    getCharacters: builder.query({
      query: (page = 1) => `/character?page=${page}`,
      providesTags: ['Character'],
    }),

    // Recuperer un personnage par son ID
    getCharacterById: builder.query({
      query: (id) => `/character/${id}`,
      providesTags: (result, error, id) => [{ type: 'Character', id }],
    }),

    // Rechercher des personnages par nom
```

```
    searchCharacters: builder.query({
      query: (name) => `/character?name=${name}`,
      providesTags: ['Character'],
    }),

    // Recuperer plusieurs personnages par leurs IDs
    getMultipleCharacters: builder.query({
      query: (ids) => `/character/${ids.join(',')}`,
      providesTags: ['Character'],
    }),
  }),
});

// Exporter les hooks generes automatiquement
export const {
  useGetCharactersQuery,
  useGetCharacterByIdQuery,
  useSearchCharactersQuery,
  useGetMultipleCharactersQuery,
} = rickAndMortyApi;
```

Listing 9 – Contenu de src/services/rickAndMortyApi.js

3.3 Étape 3 : Configurer le Store Redux avec l'API

Mettez à jour `src/store/store.js` pour intégrer l'API RTK Query :

```
import { configureStore } from '@reduxjs/toolkit';
import { rickAndMortyApi } from '../services/rickAndMortyApi';
import favoritesReducer from './favoritesSlice';

export const store = configureStore({
  reducer: {
    // Ajouter le reducer de l'API
    [rickAndMortyApi.reducerPath]: rickAndMortyApi.reducer,

    // Garder le reducer des favoris
    favorites: favoritesReducer,
  },
  // Ajouter le middleware de l'API pour activer le cache et les requetes
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(rickAndMortyApi.middleware),
});
```

Listing 10 – Contenu mis à jour de src/store/store.js

3.4 Étape 4 : Créer la Structure de Routing

Créez le fichier `src/components/Layout.jsx` pour la structure commune :

```
import { Outlet, NavLink } from 'react-router-dom';
import { useSelector } from 'react-redux';
import './Layout.css';

function Layout() {
  const favoritesCount = useSelector(
    (state) => state.favorites.favorites.length
  );

  return (
    <div className="app-layout">
      <header className="app-header">
        <h1>       Rick & Morty Explorer</h1>

        <nav className="main-nav">
          <NavLink
```

```
            to="/"
            className={({ isActive }) => isActive ? 'active' : ''}
          >
                  Accueil
          </NavLink>

          <NavLink
            to="/favorites"
            className={({ isActive }) => isActive ? 'active' : ''}
          >
                  Favoris ({favoritesCount})
          </NavLink>
        </nav>
      </header>

      <main className="app-main">
        {/* Outlet affiche le composant de la route active */}
        <Outlet />
      </main>

      <footer className="app-footer">
        <p>Données fournies par l'API Rick and Morty</p>
      </footer>
    </div>
  );
}

export default Layout;
```

Listing 11 – Contenu de src/components/Layout.jsx

3.5 Étape 5 : Créer la Page d'Accueil

Créez src/pages/HomePage.jsx :

```
import { useState } from 'react';
import { useGetCharactersQuery } from '../services/rickAndMortyApi';
import CharacterCard from '../components/CharacterCard';
import './HomePage.css';

function HomePage() {
  const [page, setPage] = useState(1);

  // Utiliser le hook genere par RTK Query
  const { data, isLoading, isError, error, isFetching } =
    useGetCharactersQuery(page);

  if (isLoading) {
    return (
      <div className="loading-container">
        <div className="spinner"></div>
        <p>Chargement des personnages...</p>
      </div>
    );
  }

  if (isError) {
    return (
      <div className="error-container">
        <h2>    Erreur</h2>
        <p>{error.message || 'Impossible de charger les personnages'}</p>
      </div>
    );
  }

  return (
```

```
    <div className="home-page">
      <div className="page-header">
        <h2>Tous les Personnages</h2>
        <p className="info-text">
          Page {data.info.pages ? `${page} sur ${data.info.pages}` : page}
        </p>
      </div>

      {isFetching && (
        <div className="fetching-indicator">
          Mise à jour...
        </div>
      )}

      <div className="characters-grid">
        {data.results.map((character) => (
          <CharacterCard key={character.id} character={character} />
        ))}
      </div>

      <div className="pagination">
        <button
          onClick={() => setPage(p => Math.max(1, p - 1))}
          disabled={page === 1 || isFetching}
        >
              Précédent
        </button>

        <span className="page-indicator">
          Page {page}
        </span>

        <button
          onClick={() => setPage(p => p + 1)}
          disabled={!data.info.next || isFetching}
        >
          Suivant
        </button>
      </div>
    </div>
  );
}

export default HomePage;
```

Listing 12 – Contenu de src/pages/HomePage.jsx

Magie de RTK Query en Action

Remarquez la simplicité du code :

- Une seule ligne pour récupérer les données : `useGetCharactersQuery(page)`
- Pas de useEffect, pas de useState pour les données
- Gestion automatique du cache : changer de page puis revenir ne refait pas l'appel
- Le flag `isFetching` distingue le premier chargement des rafraîchissements

3.6 Étape 6 : Mettre à Jour CharacterCard avec Navigation

Mettez à jour `src/components/CharacterCard.jsx` :

```
import { Link } from 'react-router-dom';
import { useSelector, useDispatch } from 'react-redux';
import { toggleFavorite } from '../store/favoritesSlice';
import './CharacterCard.css';
```

```
5
6  function CharacterCard({ character }) {
7    const dispatch = useDispatch();
8    const favorites = useSelector((state) => state.favorites.favorites);
9    const isFavorite = favorites.some((fav) => fav.id === character.id);
10
11   const handleToggleFavorite = (e) => {
12     e.preventDefault(); // Empecher la navigation lors du clic sur le bouton
13     dispatch(toggleFavorite(character));
14   };
15
16   return (
17     <Link to={`/character/${character.id}`} className="character-card-link">
18       <div className="character-card">
19         <img src={character.image} alt={character.name} />
20
21         <div className="card-content">
22           <h3>{character.name}</h3>
23           <p className="status">
24             <span className={`status-icon ${character.status.toLowerCase()}`}>
25
26             </span>
27             {character.status} - {character.species}
28           </p>
29
30           <button
31             onClick={handleToggleFavorite}
32             className={`btn-favorite ${isFavorite ? 'active' : ''}`}
33           >
34             {isFavorite ? '      ' : '      '}
35           </button>
36         </div>
37       </div>
38     </Link>
39   );
40 }
41
42 export default CharacterCard;
```

Listing 13 – Version avec navigation de CharacterCard.jsx

3.7 Étape 7 : Créer la Page de Détails

Créez `src/pages/CharacterDetailPage.jsx` :

```
1  import { useParams, useNavigate } from 'react-router-dom';
2  import { useGetCharacterByIdQuery } from '../services/rickAndMortyApi';
3  import { useSelector, useDispatch } from 'react-redux';
4  import { toggleFavorite } from '../store/favoritesSlice';
5  import './CharacterDetailPage.css';
6
7  function CharacterDetailPage() {
8    const { id } = useParams(); // Recuperer l'ID depuis l'URL
9    const navigate = useNavigate();
10   const dispatch = useDispatch();
11
12   // Appel API pour recuperer les details du personnage
13   const { data: character, isLoading, isError } =
14     useGetCharacterByIdQuery(id);
15
16   const favorites = useSelector((state) => state.favorites.favorites);
17   const isFavorite = character &&
18     favorites.some((fav) => fav.id === character.id);
19
20   if (isLoading) {
21     return (
```

```
      <div className="loading-container">
        <div className="spinner"></div>
        <p>Chargement du personnage...</p>
      </div>
    );
  }

  if (isError || !character) {
    return (
      <div className="error-container">
        <h2>    Personnage introuvable</h2>
        <button onClick={() => navigate('/')}>
          Retour à l'accueil
        </button>
      </div>
    );
  }

  return (
    <div className="character-detail-page">
      <button className="btn-back" onClick={() => navigate(-1)}>
            Retour
      </button>

      <div className="detail-container">
        <div className="detail-header">
          <img
            src={character.image}
            alt={character.name}
            className="detail-image"
          />

          <div className="detail-info">
            <h1>{character.name}</h1>

            <p className="status-badge">
              <span className={`status-icon ${character.status.toLowerCase()}`}>

              </span>
              {character.status}
            </p>

            <button
              onClick={() => dispatch(toggleFavorite(character))}
              className={`btn-favorite-large ${isFavorite ? 'active' : ''}`}
            >
              {isFavorite ? '       Retirer des favoris' : '       Ajouter aux
    favoris'}
          </button>
        </div>
      </div>

      <div className="detail-sections">
        <section className="detail-section">
          <h3>Informations Générales</h3>
          <dl>
            <dt>Espèce :</dt>
            <dd>{character.species}</dd>

            <dt>Genre :</dt>
            <dd>{character.gender}</dd>

            <dt>Origine :</dt>
            <dd>{character.origin.name}</dd>

            <dt>Dernière localisation :</dt>
```

```
              <dd>{character.location.name}</dd>
            </dl>
          </section>

          <section className="detail-section">
            <h3>Apparitions</h3>
            <p>
              Ce personnage apparaît dans{' '}
              <strong>{character.episode.length}</strong>{' '}
              épisode{character.episode.length > 1 ? 's' : ''}.
            </p>
          </section>
        </div>
      </div>
    </div>
  );
}

export default CharacterDetailPage;
```

Listing 14 – Contenu de src/pages/CharacterDetailPage.jsx

Le Pouvoir de useParams et du Cache

useParams : Récupère les paramètres dynamiques de l'URL (ici, `id`).
Cache RTK Query : Si l'utilisateur a déjà vu ce personnage dans la liste, les données sont instantanément disponibles depuis le cache, sans nouvel appel API !

3.8 Étape 8 : Créer la Page des Favoris

Créez `src/pages/FavoritesPage.jsx` :

```
import { useSelector, useDispatch } from 'react-redux';
import { Link } from 'react-router-dom';
import { removeFavorite, clearFavorites } from '../store/favoritesSlice';
import './FavoritesPage.css';

function FavoritesPage() {
  const dispatch = useDispatch();
  const favorites = useSelector((state) => state.favorites.favorites);

  const handleClearAll = () => {
    if (window.confirm('Supprimer tous les favoris ?')) {
      dispatch(clearFavorites());
    }
  };

  if (favorites.length === 0) {
    return (
      <div className="empty-favorites">
        <h2>       Aucun Favori</h2>
        <p>Vous n'avez pas encore ajouté de personnages favoris.</p>
        <Link to="/" className="btn-primary">
          Découvrir les personnages
        </Link>
      </div>
    );
  }

  return (
    <div className="favorites-page">
      <div className="page-header">
        <h2>       Mes Favoris ({favorites.length})</h2>
        <button onClick={handleClearAll} className="btn-danger">
                    Tout supprimer
```

```
        </button>
      </div>

      <div className="favorites-grid">
        {favorites.map((character) => (
          <div key={character.id} className="favorite-item">
            <Link to={`/character/${character.id}`}>
              <img src={character.image} alt={character.name} />
              <h3>{character.name}</h3>
              <p>{character.species}</p>
            </Link>

            <button
              onClick={() => dispatch(removeFavorite(character.id))}
              className="btn-remove"
            >
                  Retirer
            </button>
          </div>
        ))}
      </div>
    </div>
  );
}

export default FavoritesPage;
```

Listing 15 – Contenu de src/pages/FavoritesPage.jsx

3.9 Étape 9 : Configurer les Routes dans App.jsx

Mettez à jour `src/App.jsx` :

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Layout from './components/Layout';
import HomePage from './pages/HomePage';
import CharacterDetailPage from './pages/CharacterDetailPage';
import FavoritesPage from './pages/FavoritesPage';
import NotFoundPage from './pages/NotFoundPage';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        {/* Route parente avec Layout */}
        <Route path="/" element={<Layout />}>
          {/* Routes enfants */}
          <Route index element={<HomePage />} />
          <Route path="character/:id" element={<CharacterDetailPage />} />
          <Route path="favorites" element={<FavoritesPage />} />
          <Route path="*" element={<NotFoundPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Listing 16 – Configuration finale de App.jsx

3.10 Étape 10 : Créer la Page 404

Créez `src/pages/NotFoundPage.jsx` :

```jsx
import { Link } from 'react-router-dom';
import './NotFoundPage.css';

function NotFoundPage() {
  return (
    <div className="not-found-page">
      <h1>404</h1>
      <h2>        Dimension Inconnue</h2>
      <p>
        Cette page n'existe pas dans cette dimension.
        Peut-être existe-t-elle dans une autre ?
      </p>
      <Link to="/" className="btn-primary">
        Retour au portail principal
      </Link>
    </div>
  );
}

export default NotFoundPage;
```

Listing 17 – Contenu de src/pages/NotFoundPage.jsx

3.11 Travail à Faire par l'Étudiant (Pour le Compte Rendu)

3.11.1 Exercice 1 : Ajouter une Fonctionnalité de Recherche

Créez une nouvelle page `/search` avec :

1. Un champ de recherche pour filtrer les personnages par nom
2. Utiliser le hook `useSearchCharactersQuery`
3. Gérer le cas où aucun résultat n'est trouvé
4. Ajouter un lien de navigation vers cette page dans le Layout

Indice : Utilisez un état local pour le terme de recherche et ne lancez la requête que quand l'utilisateur a tapé au moins 3 caractères.

3.11.2 Exercice 2 : Optimiser avec le Prefetching

RTK Query permet de "pré-charger" des données avant que l'utilisateur ne navigue. Modifiez `CharacterCard` pour pré-charger les détails du personnage au survol :

```jsx
import { rickAndMortyApi } from '../services/rickAndMortyApi';

function CharacterCard({ character }) {
  const [prefetchCharacter] = rickAndMortyApi.usePrefetch('getCharacterById');

  return (
    <Link
      to={`/character/${character.id}`}
      onMouseEnter={() => prefetchCharacter(character.id)}
    >
      {/* ... contenu */}
    </Link>
  );
}
```

Testez et documentez la différence de vitesse de chargement.

3.11.3 Exercice 3 : Analyser le Cache avec Redux DevTools

1. Installez Redux DevTools dans votre navigateur
2. Naviguez dans l'application et observez l'état `rickAndMortyApi`

3. Documentez :
   — Comment les données sont stockées dans le cache
   — Combien de temps elles y restent
   — Quand elles sont invalidées

3.11.4 Exercice 4 : Ajouter la Persistance des Favoris

Intégrez `redux-persist` (vu en Séance 4) pour que les favoris persistent même après rechargement de la page. Documentez les étapes de configuration.

Question de Réflexion Finale

Question 1 : Comparez l'approche "manuelle" (useState + useEffect) et RTK Query. Pour quel type de projet privilégieriez-vous chacune ?
Question 2 : RTK Query gère automatiquement le cache. Quels sont les avantages et les inconvénients potentiels d'un cache automatique dans une application en temps réel (ex : chat, tableau de bord financier) ?
Question 3 : Proposez une architecture pour une application e-commerce complète utilisant React Router + RTK Query. Listez les routes principales et les endpoints API nécessaires.

4 Bonus : Fonctionnalités Avancées

4.1 Polling : Rafraîchissement Automatique

Pour rafraîchir automatiquement des données toutes les X secondes :

```
const { data } = useGetCharactersQuery(page, {
  pollingInterval: 30000, // Rafraichir toutes les 30 secondes
});
```

Listing 18 – Polling avec RTK Query

4.2 Retry : Nouvelle Tentative Automatique

```
const { data } = useGetCharactersQuery(page, {
  retry: 3, // Reessayer 3 fois en cas d'echec
  retryDelay: 1000, // Attendre 1 seconde entre chaque tentative
});
```

Listing 19 – Configuration des tentatives

4.3 Skip : Requête Conditionnelle

```
function UserProfile({ userId }) {
  const { data } = useGetUserQuery(userId, {
    skip: !userId, // Ne pas faire l'appel si userId est null/undefined
  });

  // ...
}
```

Listing 20 – Ne lancer une requête que si une condition est remplie

5 Conclusion

Dans cette séance, nous avons appris à :

1. Créer une architecture multi-pages avec React Router
2. Gérer la navigation avec des liens et programmatiquement
3. Utiliser les paramètres d'URL pour les pages dynamiques
4. Simplifier drastiquement les appels API avec RTK Query
5. Bénéficier du cache automatique et du partage de données
6. Gérer élégamment le chargement et les erreurs

Points clés à retenir :

— React Router synchronise l'URL avec vos composants
— RTK Query réduit le code boilerplate de 70-80%
— Le cache automatique améliore drastiquement les performances
— L'invalidation par tags garantit la fraîcheur des données
— Cette architecture est scalable pour de grandes applications