Architecture Hexagonale Simplifiée pour React RUCH

Cette documentation présente une architecture hexagonale simplifiée pour les projets React, organisée de manière intuitive respectant une structure ressemblant a une structure classique en React, en utilisant des contexts ainsi que React Query, avec les tests placés à côté des fichiers qu'ils testent.

Table des matières

Structure du projet

La structure proposée conserve l'organisation classique d'un projet React tout en intégrant les principes de l'architecture hexagonale par domaines métier:

```
src/
├── components/              # Composants React partagés
│   ├── ui/                  # Composants d'UI réutilisables
│   │   ├── Button.tsx
│   │   ├── Button.test.tsx      # Test colocalisé avec son composant
│   │   └── ...
│   └── layouts/             # Layouts partagés
```

```
│
├── pages/                  # Pages React (composants de routes)
│   ├── Home.tsx
│   ├── Home.test.tsx          # Test colocalisé avec sa page
│   └── ...
│
├── hooks/                  # Hooks React partagés
│   ├── useForm.ts
│   ├── useForm.test.ts        # Test colocalisé avec son hook
│   └── ...
│
├── domain/                  # Tous les domaines métier
│   ├── user/                # Domaine utilisateur
│   │   ├── entities/          # Modèles de données
│   │   │   ├── User.ts
│   │   │
│   │   ├── ports/             # Interfaces pour les adaptateurs
│   │   │   └── UserPorts.ts
│   │   │
│   │   ├── services/          # Services métier
│   │   │   ├── UserService.ts
│   │   │   └── UserService.test.ts  # Tests des services
│   │   │
│   │   ├── adapters/          # Implémentations concrètes
│   │   │   ├── UserApiAdapter.ts
│   │   │   └── UserApiAdapter.test.ts  # Tests des adaptateurs
│   │   │
│   │   └── hooks/             # Hooks spécifiques au domaine
│   │       ├── useUser.ts
│   │       └── useUser.test.ts  # Tests des hooks
│   │   └── ui/                # Ui spécifiques au domaine
│   │       ├── UserView.ts
│   │       └── userView.test.ts  # Tests des vues
│   │
│   └── product/               # Autre domaine
│       ├── entities/
│       ├── ports/
│       ├── services/
```

```
│       ├── adapters/
│       └── hooks/
│
├── context/                  # Contextes React globaux
│   ├── ServiceContext.tsx
│   └── ServiceContext.test.tsx  # Tests du contexte
│
├── utils/                    # Utilitaires et helpers
│   ├── formatters.ts
│   ├── formatters.test.ts       # Tests des utilitaires
│   └── ...
│
├── mocks/                    # Configuration globale MSW
│   ├── handlers/             # Handlers organisés par domaine
│   │   ├── user.ts           # Handlers pour le domaine user
│   │   └── product.ts          # Handlers pour le domaine product
│   ├── server.ts             # Serveur MSW pour Node.js
│   └── browser.ts            # Worker MSW pour le navigateur
│
├── assets/                   # Ressources statiques
│   ├── images/
│   └── styles/
│
├── App.tsx                   # Point d'entrée de l'application
├── App.test.tsx              # Test du composant App
├── index.tsx                 # Fichier d'initialisation
└── setupTests.ts             # Configuration globale des tests
```

Organisation par domaines

Chaque domaine est organisé selon les principes de l'architecture hexagonale, avec les tests colocalisés:

Exemple pour le domaine "user"

```
domain/
└── user/
    ├── entities/             # Modèles de données
```

```
    │   ├── User.ts              # Interface User
    │   └── User.test.ts         # Tests de validation des entités
    │
    ├── ports/                   # Interfaces pour les adaptateurs
    │   └── UserRepository.ts    # Interface pour accéder aux données
    │
    ├── services/                # Services métier
    │   ├── UserService.ts       # Logique métier utilisateur
    │   └── UserService.test.ts  # Tests des services
    │
    ├── adapters/                # Implémentations concrètes
    │   ├── UserApiAdapter.ts    # Accès à l'API REST
    │   └── UserApiAdapter.test.ts  # Tests de l'adaptateur
    │
    └── hooks/                   # Hooks React spécifiques
        ├── useUser.ts           # Hook combinant service et React Query
        └── useUser.test.ts      # Tests du hook
```

Exemples d'implémentation

1. Définition des entités

```
// src/domain/user/entities/User.ts
export interface User {
  id: string;
  name: string;
  email: string;
  role: 'admin' | 'user';
  createdAt: string;
}
```

2. Définition des ports (interfaces)

```
// src/domain/user/ports/UserPorts.ts
import { User } from '../entities/User';

export interface UserPorts {
```

```
  getUser(id: string): Promise<User>;
  updateUser(user: User): Promise<User>;
  getUserList(): Promise<User[]>;
}
```

3. Services métier

```
// src/domain/user/services/UserService.ts
import { User } from '../entities/User';
import { UserPorts } from '../ports/UserPorts';

export class UserService {
  constructor(private userPorts: UserPorts) {}

  async getUser(id: string): Promise<User> {
    return this.userPorts.getUser(id);
  }

  async updateUser(user: User): Promise<User> {
    return this.userPorts.updateUser(user);
  }

  async getUserList(): Promise<User[]> {
    return this.userPorts.getUserList();
  }

  // Exemple de logique métier plus complexe
  async promoteToAdmin(userId: string): Promise<User> {
    const user = await this.getUser(userId);
    if (user.role === 'admin') {
      throw new Error('User is already an admin');
    }

    const updatedUser = { ...user, role: 'admin' as const };
    return this.updateUser(updatedUser);
  }
}
```

4. Adaptateurs concrets

```
// src/domain/user/adapters/UserApiAdapter.ts
import { User } from '../entities/User';
import { UserPorts } from '../ports/UserPorts';

export class UserApiAdapter implements UserPorts {
  private baseUrl: string;

  constructor(baseUrl: string = '/api/users') {
    this.baseUrl = baseUrl;
  }

  async getUser(id: string): Promise<User> {
    const response = await fetch(`${this.baseUrl}/${id}`);

    if (!response.ok) {
      throw new Error(`Error fetching user: ${response.statusText}`);
    }

    return await response.json();
  }

  async updateUser(user: User): Promise<User> {
    const response = await fetch(`${this.baseUrl}/${user.id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(user)
    });

    if (!response.ok) {
      throw new Error(`Error updating user: ${response.statusText}`);
    }

    return await response.json();
  }

  async getUserList(): Promise<User[]> {
```

```
    const response = await fetch(`${this.baseUrl}/`);

    if (!response.ok) {
      throw new Error(`Error fetching users: ${response.statusText}`);
    }

    return await response.json();
  }
}
```

5. Contexte pour l'injection de dépendances

```
// src/context/ServiceContext.tsx
import React, { createContext, useContext } from 'react';

// Import des services et adaptateurs
import { UserService } from '../domain/user/services/UserService';
import { UserApiAdapter } from '../domain/user/adapters/UserApiAdapter';
import { ProductService } from '../domain/product/services/ProductServic
e';
import { ProductApiAdapter } from '../domain/product/adapters/ProductApi
Adapter';

// Configuration de base
const API_BASE_URL = process.env.REACT_APP_API_BASE_URL || '/api';

// Création des adaptateurs
const userRepository = new UserApiAdapter(API_BASE_URL);
const productRepository = new ProductApiAdapter(API_BASE_URL);

// Création des services avec leurs dépendances
const userService = new UserService(userRepository);
const productService = new ProductService(productRepository);

// Type pour les services
interface Services {
  userService: UserService;
  productService: ProductService;
```

```
}

// Services par défaut
const defaultServices: Services = {
  userService,
  productService
};

// Création du contexte
const ServiceContext = createContext<Services>(defaultServices);

// Provider pour injection de dépendances
export const ServiceProvider: React.FC<{
  services?: Partial<Services>;
  children: React.ReactNode;
}> = ({ services, children }) => {
  const value = { ...defaultServices, ...services };

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

// Hook pour utiliser les services
export const useServices = () => useContext(ServiceContext);
```

6. Hook spécifique au domaine utilisant React Query

```
// src/domain/user/hooks/useUser.ts
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-q
uery';
import { User } from '../entities/User';
import { useServices } from '../../../context/ServiceContext';

// Clés pour React Query
export const USER_QUERY_KEYS = {
```

```
  all: ['users'] as const,
  lists: () => [...USER_QUERY_KEYS.all, 'list'] as const,
  detail: (id: string) => [...USER_QUERY_KEYS.all, 'detail', id] as const,
};

export function useUser(userId: string) {
  const { userService } = useServices();
  const queryClient = useQueryClient();

  // Requête pour obtenir l'utilisateur
  const userQuery = useQuery({
    queryKey: USER_QUERY_KEYS.detail(userId),
    queryFn: () => userService.getUser(userId),
    enabled: !!userId,
  });

  // Mutation pour mettre à jour l'utilisateur
  const updateUserMutation = useMutation({
    mutationFn: (updatedUser: User) => userService.updateUser(updatedUs
er),
    onSuccess: (updatedUser) => {
      queryClient.setQueryData(USER_QUERY_KEYS.detail(userId), updatedU
ser);
      queryClient.invalidateQueries({ queryKey: USER_QUERY_KEYS.lists() });
    },
  });

  // Mutation pour promouvoir un utilisateur en admin
  const promoteToAdminMutation = useMutation({
    mutationFn: (id: string) => userService.promoteToAdmin(id),
    onSuccess: (updatedUser) => {
      queryClient.setQueryData(USER_QUERY_KEYS.detail(userId), updatedU
ser);
      queryClient.invalidateQueries({ queryKey: USER_QUERY_KEYS.lists() });
    },
  });

  return {
```

```
    user: userQuery.data,
    isLoading: userQuery.isLoading,
    error: userQuery.error,
    updateUser: updateUserMutation.mutate,
    isUpdating: updateUserMutation.isPending,
    promoteToAdmin: promoteToAdminMutation.mutate,
    isPromoting: promoteToAdminMutation.isPending,
  };
}
```

Tests colocalisés

1. Test d'un service

```
// src/domain/user/services/UserService.test.ts
import { UserService } from './UserService';
import { UserPorts } from '../ports/UserPorts';
import { User } from '../entities/User';

// Mock du Ports
const mockUser: User = {
  id: '1',
  name: 'Test User',
  email: 'test@example.com',
  role: 'user',
  createdAt: '2023-01-01T00:00:00Z'
};

const mockUserRepository: UserRepository = {
  getUser: jest.fn().mockResolvedValue(mockUser),
  updateUser: jest.fn().mockImplementation((user) => Promise.resolve(use
r)),
  getUserList: jest.fn().mockResolvedValue([mockUser]),
};

describe('UserService', () => {
  let userService: UserService;
```

```
  beforeEach(() => {
    jest.clearAllMocks();
    userService = new UserService(mockUserRepository);
  });

  it('should get a user by id', async () => {
    const user = await userService.getUser('1');
    expect(user).toEqual(mockUser);
    expect(mockUserRepository.getUser).toHaveBeenCalledWith('1');
  });

  it('should update a user', async () => {
    const updatedUser = { ...mockUser, name: 'Updated Name' };
    const result = await userService.updateUser(updatedUser);

    expect(result).toEqual(updatedUser);
    expect(mockUserRepository.updateUser).toHaveBeenCalledWith(update
dUser);
  });

  it('should promote a user to admin', async () => {
    const userId = '1';
    await userService.promoteToAdmin(userId);

    // Vérifier que getUser a été appelé
    expect(mockUserRepository.getUser).toHaveBeenCalledWith(userId);

    // Vérifier que updateUser a été appelé avec le bon rôle
    expect(mockUserRepository.updateUser).toHaveBeenCalledWith({
      ...mockUser,
      role: 'admin'
    });
  });

  it('should throw error when promoting an admin', async () => {
    const adminUser = { ...mockUser, role: 'admin' };
    mockUserRepository.getUser = jest.fn().mockResolvedValue(adminUse
r);
```

```
    await expect(userService.promoteToAdmin('1')).rejects.toThrow('User is
already an admin');
  });
});
```

2. Test d'un adaptateur API avec MSW

```
// src/domain/user/adapters/UserApiAdapter.test.ts
import { UserApiAdapter } from './UserApiAdapter';
import { server } from '../../../mocks/server';
import { http, HttpResponse } from 'msw';

// Réinitialiser les handlers MSW pour ces tests spécifiques
beforeEach(() => {
  server.resetHandlers();
});

describe('UserApiAdapter', () => {
  const baseUrl = '/api';
  let adapter: UserApiAdapter;

  beforeEach(() => {
    adapter = new UserApiAdapter(baseUrl);
  });

  it('should fetch a user by id', async () => {
    // Configurer le handler MSW pour ce test
    server.use(
      http.get(`${baseUrl}/users/1`, () => {
        return HttpResponse.json({
          id: '1',
          name: 'John Doe',
          email: 'john@example.com',
          role: 'user',
          createdAt: '2023-01-01T00:00:00Z'
        });
      })
```

```
    );

    const user = await adapter.getUser('1');

    expect(user).toBeDefined();
    expect(user.id).toBe('1');
    expect(user.name).toBe('John Doe');
  });

  it('should handle error when fetching user', async () => {
    // Simuler une erreur
    server.use(
      http.get(`${baseUrl}/users/999`, () => {
        return new HttpResponse(null, { status: 404 });
      })
    );

    await expect(adapter.getUser('999')).rejects.toThrow('Error fetching use
r');
  });

  it('should update a user', async () => {
    const updatedUser = {
      id: '1',
      name: 'Updated Name',
      email: 'john@example.com',
      role: 'user',
      createdAt: '2023-01-01T00:00:00Z'
    };

    server.use(
      http.put(`${baseUrl}/users/1`, async ({ request }) => {
        const body = await request.json();
        return HttpResponse.json({ ...body, updatedAt: '2023-02-01T00:00:00
Z' });
      })
    );
```

```
    const result = await adapter.updateUser(updatedUser);

    expect(result).toBeDefined();
    expect(result.name).toBe('Updated Name');
    expect(result.updatedAt).toBe('2023-02-01T00:00:00Z');
  });
});
```

3. Test d'un hook avec React Testing Library

```
// src/domain/user/hooks/useUser.test.ts
import { renderHook, act, waitFor } from '@testing-library/react';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import { useUser } from './useUser';
import { ServiceProvider } from '../../../context/ServiceContext';
import { UserService } from '../services/UserService';
import { User } from '../entities/User';

// Mock du service utilisateur
const mockUser: User = {
  id: '1',
  name: 'Test User',
  email: 'test@example.com',
  role: 'user',
  createdAt: '2023-01-01T00:00:00Z'
};

const mockUserService = {
  getUser: jest.fn().mockResolvedValue(mockUser),
  updateUser: jest.fn().mockImplementation((user: User) => Promise.resolve
(user)),
  promoteToAdmin: jest.fn().mockImplementation(() =>
    Promise.resolve({ ...mockUser, role: 'admin' })
  ),
} as unknown as UserService;

// Wrapper avec providers pour les tests a mettre dans un utils
const createWrapper = () => {
```

```
  const queryClient = new QueryClient({
    defaultOptions: {
      queries: {
        retry: false,
      },
    },
  });

  return ({ children }) => (
    <QueryClientProvider client={queryClient}>
      <ServiceProvider services={{ userService: mockUserService }}>
        {children}
      </ServiceProvider>
    </QueryClientProvider>
  );
};

describe('useUser hook', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('should fetch user data', async () => {
    const { result } = renderHook(() => useUser('1'), {
      wrapper: createWrapper(),
    });

    // Initialement loading doit être true
    expect(result.current.isLoading).toBe(true);

    // Attendre que les données soient chargées
    await waitFor(() => {
      expect(result.current.isLoading).toBe(false);
    });

    // Vérifier les données
    expect(result.current.user).toEqual(mockUser);
    expect(mockUserService.getUser).toHaveBeenCalledWith('1');
```

```
  });

  it('should update user data', async () => {
    const { result } = renderHook(() => useUser('1'), {
      wrapper: createWrapper(),
    });

    // Attendre le chargement initial
    await waitFor(() => {
      expect(result.current.isLoading).toBe(false);
    });

    // Mettre à jour l'utilisateur
    const updatedUser = { ...mockUser, name: 'Updated Name' };

    act(() => {
      result.current.updateUser(updatedUser);
    });

    // Vérifier que la mise à jour a été appelée
    expect(mockUserService.updateUser).toHaveBeenCalledWith(updatedU
ser);

    // Attendre la fin de la mise à jour
    await waitFor(() => {
      expect(result.current.isUpdating).toBe(false);
    });
  });

  it('should promote user to admin', async () => {
    const { result } = renderHook(() => useUser('1'), {
      wrapper: createWrapper(),
    });

    // Attendre le chargement initial
    await waitFor(() => {
      expect(result.current.isLoading).toBe(false);
    });
```

```
    // Promouvoir l'utilisateur
    act(() => {
      result.current.promoteToAdmin('1');
    });

    // Vérifier que la promotion a été appelée
    expect(mockUserService.promoteToAdmin).toHaveBeenCalledWith('1');

    // Attendre la fin de la promotion
    await waitFor(() => {
      expect(result.current.isPromoting).toBe(false);
    });
  });
});
```

Intégration avec React Query

Composant utilisant le hook de domaine

```
// src/pages/UserProfilePage.tsx
import React from 'react';
import { useParams } from 'react-router-dom';
import { useUser } from '../domain/user/hooks/useUser';

export const UserProfilePage: React.FC = () => {
  const { userId } = useParams<{ userId: string }>();
  const {
    user,
    isLoading,
    error,
    updateUser,
    isUpdating,
    promoteToAdmin,
    isPromoting
  } = useUser(userId || '');

  if (isLoading) return <div>Chargement...</div>;
```

```
if (error) return <div>Erreur: {error.message}</div>;
if (!user) return <div>Utilisateur non trouvé</div>;

const handleNameChange = () => {
  const newName = prompt('Nouveau nom:', user.name);
  if (newName && newName !== user.name) {
    updateUser({ ...user, name: newName });
  }
};

const handlePromoteToAdmin = () => {
  if (window.confirm('Promouvoir cet utilisateur en administrateur?')) {
    promoteToAdmin(user.id);
  }
};

return (
  <div className="user-profile">
    <h1>Profil Utilisateur</h1>

    <div className="user-details">
      <h2>{user.name}</h2>
      <p>Email: {user.email}</p>
      <p>Rôle: {user.role}</p>
      <p>Créé le: {new Date(user.createdAt).toLocaleDateString()}</p>
    </div>

    <div className="user-actions">
      <button
        onClick={handleNameChange}
        disabled={isUpdating}
      >
        {isUpdating ? 'Modification en cours...' : 'Modifier le nom'}
      </button>

      {user.role !== 'admin' && (
        <button
          onClick={handlePromoteToAdmin}
```

```
          disabled={isPromoting}
        >
          {isPromoting ? 'Promotion en cours...' : 'Promouvoir en admin'}
        </button>
      )}
    </div>
  </div>
 );
};

// src/pages/UserProfilePage.test.tsx - Test colocalisé
import { render, screen, waitFor, fireEvent } from '@testing-library/react';
import { MemoryRouter, Routes, Route } from 'react-router-dom';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import { ServiceProvider } from '../context/ServiceContext';
import { UserProfilePage } from './UserProfilePage';
import { UserService } from '../domain/user/services/UserService';

// Mocks similaires à ceux du test du hook
// ...

describe('UserProfilePage', () => {
 // Tests du composant
 // ...
});
```

Configuration MSW

1. Handlers par domaine

```
// src/mocks/handlers/user.ts
import { http, HttpResponse } from 'msw';

// Base de données mockée
const users = {
 '1': {
   id: '1',
   name: 'John Doe',
```

```
    email: 'john@example.com',
    role: 'user',
    createdAt: '2023-01-01T00:00:00Z'
  },
  '2': {
    id: '2',
    name: 'Jane Smith',
    email: 'jane@example.com',
    role: 'admin',
    createdAt: '2023-01-02T00:00:00Z'
  }
};

export const userHandlers = [
  // GET /api/users
  http.get('/api/users', () => {
    return HttpResponse.json(Object.values(users));
  }),

  // GET /api/users/:id
  http.get('/api/users/:id', ({ params }) => {
    const { id } = params;

    if (!users[id as string]) {
      return new HttpResponse(null, { status: 404 });
    }

    return HttpResponse.json(users[id as string]);
  }),

  // PUT /api/users/:id
  http.put('/api/users/:id', async ({ params, request }) => {
    const { id } = params;

    if (!users[id as string]) {
      return new HttpResponse(null, { status: 404 });
    }
```

```
    const updateData = await request.json();
    users[id as string] = { ...users[id as string], ...updateData };

    return HttpResponse.json(users[id as string]);
  })
];
```

2. Configuration du serveur MSW

```
// src/mocks/server.ts
import { setupServer } from 'msw/node';
import { userHandlers } from './handlers/user';
import { productHandlers } from './handlers/product';

// Création du serveur MSW avec tous les handlers
export const server = setupServer(
  ...userHandlers,
  ...productHandlers
);
```

3. Configuration du worker MSW pour le navigateur

```
// src/mocks/browser.ts
import { setupWorker } from 'msw/browser';
import { userHandlers } from './handlers/user';
import { productHandlers } from './handlers/product';

// Création du worker MSW avec tous les handlers
export const worker = setupWorker(
  ...userHandlers,
  ...productHandlers
);
```

4. Configuration globale des tests

```
// src/setupTests.ts
import '@testing-library/jest-dom';
```

```
import { server } from './mocks/server';

// Activer le serveur MSW avant tous les tests
beforeAll(() => server.listen({ onUnhandledRequest: 'warn' }));

// Réinitialiser les handlers entre les tests
afterEach(() => server.resetHandlers());

// Fermer le serveur après tous les tests
afterAll(() => server.close());
```

5. Utilisation de MSW en développement

```
// src/index.tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

async function bootstrap() {
  // Activer MSW en développement uniquement
  if (process.env.NODE_ENV === 'development') {
    const { worker } = await import('./mocks/browser');
    await worker.start({ onUnhandledRequest: 'bypass' });
  }

  const root = ReactDOM.createRoot(document.getElementById('root') as H
TMLElement);

  root.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>
  );
}

bootstrap();
```

Conclusion

Cette architecture présente plusieurs avantages:

1. **Organisation intuitive**:
   - Code organisé par domaines métier
   - Tests colocalisés avec le code qu'ils testent
   - Structure familière pour les développeurs React
2. **Séparation des préoccupations**:
   - Logique métier indépendante de l'interface utilisateur
   - Adaptateurs pour les sources de données externes
   - Hooks spécifiques aux domaines pour connecter le tout
3. **Testabilité améliorée**:
   - Tests plus faciles à écrire et à maintenir
   - MSW pour mocker les API sans mocking manuel
   - Injection de dépendances via React Context
4. **Évolutivité**:
   - Facile d'ajouter de nouveaux domaines
   - Possibilité de changer l'implémentation des adaptateurs
   - Structure qui s'adapte à la taille du projet

Cette approche reste simple tout en offrant les avantages de l'architecture hexagonale, permettant de créer des applications React maintenables et testables.