### **Cahier des Charges Technique et d'Implémentation Front-End - SkillForge AI**

* **Version :** 1.0
* **Date :** 12 juin 2025
* **Propriétaire :** Lead Développeur Front-End

### **Partie 1 : Introduction et Principes Directeurs**

#### **1.1. Objectif de ce Document**

Ce document est le **guide de construction et la référence unique** pour l'intégralité de l'application cliente (Front-End) de SkillForge AI. Sa mission est de traduire les exigences fonctionnelles et les maquettes UX/UI en une application web de haute qualité, en fournissant à l'équipe de développement un ensemble de règles, de conventions et de patrons d'architecture clairs et prescriptifs.

L'objectif final est de garantir la livraison d'une Single-Page Application (SPA) qui soit :

* **Fonctionnelle :** Implémentant fidèlement toutes les fonctionnalités décrites dans les spécifications.
* **Réactive :** Offrant une expérience utilisateur fluide, rapide et sans temps de latence perçu.
* **Robuste :** Gérant proprement les états (chargement, erreur, succès) et les interactions complexes.
* **Accessible :** Conçue en suivant les meilleures pratiques d'accessibilité web (WCAG).
* **Maintenable :** Écrite de manière propre et modulaire pour faciliter les itérations futures et l'évolution du produit.

#### **1.2. Audience Cible**

Ce CDC est destiné aux profils suivants :

* **Audience Primaire :**
  + **Développeurs Front-End :** Il s'agit de leur manuel de référence pour l'architecture, les conventions de code, la gestion de l'état et l'implémentation des composants.
* **Audience Secondaire :**
  + **Designers UX/UI :** Pour comprendre le cadre technique, la bibliothèque de composants et les contraintes dans lesquelles leurs designs prendront vie, assurant une collaboration fluide.
  + **Chefs de Produit :** Pour appréhender la structure technique de la partie visible de l'application et l'organisation des fonctionnalités.
  + **Ingénieurs QA (Assurance Qualité) :** Pour comprendre l'architecture des composants et les flux de données afin de concevoir des plans de test pertinents et efficaces.

#### **1.3. Articulation avec les Autres Documents**

Ce document est un "Spoke" (rayon) spécialisé qui s'inscrit dans l'écosystème documentaire du projet piloté par le "Hub" (moyeu), le Guide d'Architecture Générale (GAG). Il ne duplique pas l'information mais détaille son domaine de responsabilité.

* **Relation avec le Guide d'Architecture Générale (GAG) :**
  + Le **GAG** décrète que le Front-End est une "Single-Page Application utilisant React".
  + Ce **CDC Front-End** détaille l'**architecture interne** de cette SPA : comment nous structurons nos dossiers, gérons notre état, et organisons nos composants.
* **Relation avec les Spécifications Fonctionnelles & User Stories :**
  + Le document de **User Stories** définit **CE QUE** l'application doit permettre à l'utilisateur de faire.
  + Ce **CDC Front-End** définit **COMMENT** ces fonctionnalités sont techniquement construites côté client.
* **Relation avec la Spécification OpenAPI du Back-End :**
  + La spécification **OpenAPI (openapi.json)** est le **contrat de données** immuable entre le Front-End et le Back-End. Elle est la source de vérité absolue pour les endpoints, les formats de requêtes et de réponses.
  + Ce **CDC Front-End** ne redéfinit jamais ce contrat. Il prescrit les **outils et les méthodes** que nous utilisons pour consommer ce contrat de manière sûre et typée (ex: client Axios, génération de types TypeScript).
* **Relation avec le CDC Back-End :**
  + Il s'agit d'un document "frère" qui opère en parallèle. Aucune information sur l'implémentation interne du Back-End (sa base de données, sa logique métier interne) n'est pertinente ou autorisée dans ce document. La seule frontière et le seul point de contact est l'API REST, gouvernée par la spécification OpenAPI.

### **Partie 2 : Architecture Technique Front-End**

Cette section est le plan de construction de notre application React. Elle définit la stack technologique, l'organisation du code et la stratégie de gestion des données.

#### **2.1. Stack Technologique Officielle**

La stack suivante est le standard unique et non-négociable pour le développement Front-End de SkillForge AI. Le choix de chaque outil a été fait pour sa modernité, sa performance et la robustesse de son écosystème.

| Domaine | Technologie | Version | Justification |
| --- | --- | --- | --- |
| Framework | React | 18+ | L'écosystème le plus mature pour la construction d'interfaces utilisateur complexes via une approche déclarative et basée sur les composants. Utilisation exclusive des Hooks. |
| Langage | TypeScript | 5+ | Indispensable pour la sécurité des types, la maintenabilité à grande échelle et une auto-complétion intelligente, réduisant drastiquement les bugs. |
| Outil de Build | Vite | 5+ | Pour son serveur de développement ultra-rapide (HMR natif) et ses performances de build optimisées, offrant une expérience de développement supérieure. |
| Style | Tailwind CSS | 3+ | Pour une approche "utility-first" qui permet de construire des designs complexes rapidement et de manière cohérente, sans quitter le balisage JSX. |
| Routage | React Router | 6+ | La bibliothèque standard de-facto pour la gestion des routes dans une Single-Page Application React. |
| Client HTTP | Axios | 1.x+ | Un client HTTP robuste et éprouvé, facilitant les requêtes vers l'API et la gestion des intercepteurs (pour l'authentification et les erreurs). |
| Gestion d'État Global | Zustand | 4.x+ | Une bibliothèque de gestion d'état minimaliste, simple et puissante, qui évite le "boilerplate" lourd d'autres solutions. |
| Gestion d'État Serveur | TanStack Query | 5+ | Outil incontournable pour la gestion des données asynchrones. Il gère le fetching, la mise en cache, la synchronisation et la mise à jour des données serveur. |
| Tests | Vitest & React Testing Library | Dernières | Un couple moderne et performant pour les tests, parfaitement intégré avec Vite, qui favorise des tests centrés sur le comportement utilisateur. |

#### **2.2. Structure des Dossiers**

Pour assurer la cohérence et la scalabilité du projet, nous adoptons une approche **"Feature-Sliced Design" (adaptée)**. Les fonctionnalités sont isolées dans leurs propres répertoires, favorisant une forte cohésion et un faible couplage.

La structure racine du code (src/) doit impérativement suivre le modèle suivant :

src/

├── App.tsx # Composant racine, assemblage des providers et du routeur

├── main.tsx # Point d'entrée de l'application

├── assets/ # Ressources statiques (images, polices, icônes)

├── components/

│ └── ui/ # Composants UI partagés, réutilisables et "bêtes" (Button, Card...)

├── config/ # Fichiers de configuration (instance Axios, constantes...)

├── features/ # Répertoires des fonctionnalités métier (ex: authentification)

│ └── auth/

│ ├── api/ # Fonctions/hooks d'appels API spécifiques à l'auth

│ ├── components/ # Composants spécifiques à l'auth (LoginForm...)

│ └── index.ts # Point d'export public de la feature

├── hooks/ # Hooks React partagés et réutilisables (ex: useDebounce)

├── lib/ # Fonctions utilitaires non liées à React (ex: formatDate)

├── pages/ # Composants de haut niveau qui représentent une page/route

├── providers/ # Fournisseurs de contexte React (ex: AuthProvider)

├── stores/ # Stores globaux Zustand (ex: useAuthStore.ts)

└── types/ # Interfaces et types TypeScript partagés

**Rôle de chaque dossier principal :**

* **components/ui :** Notre Design System. Contient des composants génériques sans logique métier.
* **features :** Le cœur de l'application. Chaque sous-dossier est une fonctionnalité autonome.
* **pages :** La couche d'assemblage. Un composant de page importe plusieurs composants de features pour construire une vue complète correspondant à une route.
* **stores :** Contient la définition de nos "slices" d'état global avec Zustand.

#### **2.3. Stratégie de Gestion de l'État (State Management)**

Une gestion de l'état claire est cruciale. Voici la hiérarchie à respecter scrupuleusement :

1. **État Local (Component State)**
   * **Outils :** useState, useReducer de React.
   * **Quand l'utiliser :** Pour tout état qui n'a pas besoin d'être connu en dehors d'un composant et de ses enfants directs. Exemples : l'état d'ouverture/fermeture d'une modale, la valeur d'un champ de formulaire non contrôlé.
2. **État Global Partagé (Global UI State)**
   * **Outil Obligatoire :** **Zustand**.
   * **Quand l'utiliser :** Pour l'état qui doit être accessible et modifiable depuis des endroits déconnectés de l'arbre des composants.
   * **Exemples Concrets :**
     + Le statut d'authentification de l'utilisateur (isLoading, isAuthenticated).
     + Les informations du profil de l'utilisateur connecté.
     + L'état d'une notification globale (Toast).
   * **Stores Initiaux :** Le premier store à créer sera stores/useAuthStore.ts pour gérer l'état de l'utilisateur.
3. **État Serveur (Server Cache State)**
   * **Outil Obligatoire :** **TanStack Query (React Query)**.
   * **Règle Absolue :** Toute donnée provenant d'une requête au Back-End DOIT être gérée via TanStack Query. Il est interdit de stocker des données serveur dans un useState ou un store Zustand.
   * **Justification :** TanStack Query n'est pas un simple "fetcher". Il gère pour nous :
     + La mise en cache intelligente des données.
     + L'invalidation automatique du cache.
     + Le re-fetching automatique des données (au focus de la fenêtre, à la reconnexion...).
     + Les états de chargement (isLoading) et d'erreur (isError) de manière déclarative.
     + La pagination et le défilement infini.
   * L'utilisation de cet outil élimine des centaines de lignes de code useEffect complexes et prévient d'innombrables bugs liés à la synchronisation des données.

### **Partie 3 : Design System et Bibliothèque de Composants**

Cette partie est dédiée à la construction des briques élémentaires de notre interface. L'objectif est de créer un ensemble de composants réutilisables, cohérents et de haute qualité qui accéléreront le développement et garantiront une expérience utilisateur homogène sur toute la plateforme.

#### **3.1. Philosophie du Design System**

Nous adoptons une approche inspirée de l'**"Atomic Design"**. Cela signifie que nous construisons notre interface en assemblant des pièces de plus en plus complexes, ce qui garantit la modularité et la réutilisabilité.

Notre système se décompose en trois niveaux principaux :

1. **Atomes :** Les éléments les plus basiques de notre UI, qui ne peuvent être décomposés davantage (ex: une couleur, une police de caractère, une icône). Ces "atomes" sont définis comme des "design tokens" dans notre fichier de configuration Tailwind CSS.
2. **Molécules :** Des groupes d'atomes fonctionnels qui forment des unités de base de notre interface. Ce sont nos **composants d'UI fondamentaux**, stockés dans /components/ui/. Ils sont "bêtes" (agnostiques de la logique métier). Exemple : un champ de formulaire composé d'un label (atome), d'un input (atome) et d'un message d'erreur (atome).
3. **Organismes :** Des assemblages de molécules qui forment des sections distinctes et contextuelles de l'interface. Ces composants plus complexes, qui peuvent contenir de la logique métier, sont situés dans les répertoires de features/. Exemple : un formulaire de connexion (<LoginForm />) composé de plusieurs molécules de champs de formulaire et d'un bouton.

#### **3.2. Catalogue des Composants d'UI Fondamentaux (/components/ui)**

La liste suivante définit les composants "moléculaires" de base à construire en priorité. Chaque composant doit être développé en isolation, être entièrement typé (TypeScript) et testé.

* **Nom du Composant :** <Button />  
  + **Description :** Le composant principal pour toutes les actions utilisateur cliquables.
  + **Props Principales :** variant ('primary', 'secondary', 'danger'), size ('sm', 'md', 'lg'), onClick, isLoading (affiche un spinner), disabled, leftIcon, rightIcon.
  + **États :** Doit gérer visuellement les états hover, focus, active, et disabled.
* **Nom du Composant :** <Input />  
  + **Description :** Champ de saisie pour le texte, les mots de passe, les e-mails, etc.
  + **Props Principales :** type, label, placeholder, value, onChange, error (affiche un message d'erreur), disabled, leftIcon.
* **Nom du Composant :** <Textarea />  
  + **Description :** Identique à <Input /> mais pour la saisie de texte sur plusieurs lignes.
  + **Props Principales :** label, placeholder, value, onChange, error, disabled, rows.
* **Nom du Composant :** <Card />  
  + **Description :** Un conteneur stylisé avec une ombre et des coins arrondis, servant de base pour afficher des blocs de contenu (ex: une carte de projet).
  + **Props Principales :** children, padding ('sm', 'md', 'lg').
* **Nom du Composant :** <Modal />  
  + **Description :** Une fenêtre de dialogue qui s'affiche par-dessus le contenu principal de la page.
  + **Props Principales :** isOpen, onClose, title, children (pour le corps), footer (pour les boutons d'action).
* **Nom du Composant :** <Spinner />  
  + **Description :** Une animation indiquant un état de chargement.
  + **Props Principales :** size ('sm', 'md', 'lg'), color.
* **Nom du Composant :** <Alert />  
  + **Description :** Un message contextuel pour fournir un feedback à l'utilisateur (ex: succès, information, avertissement, erreur).
  + **Props Principales :** status ('success', 'info', 'warning', 'error'), title, description, isClosable.
* **Nom du Composant :** <Avatar />  
  + **Description :** Affiche l'image de profil d'un utilisateur ou ses initiales en fallback.
  + **Props Principales :** src (URL de l'image), name (pour générer les initiales), size.
* **Nom du Composant :** <Badge /> / <Tag />  
  + **Description :** Un petit label pour afficher des statuts ou des mots-clés (ex: statut d'un projet, compétence technique).
  + **Props Principales :** variant (couleur), children.

#### **3.3. Theming et Conventions de Style**

1. **Technologie Exclusive : Tailwind CSS**
   * **Règle :** Tout le style de l'application doit être implémenté via les classes utilitaires de Tailwind CSS. L'écriture de fichiers CSS personnalisés (.css, .scss) est interdite, sauf pour des cas très spécifiques (ex: styles de base globaux dans index.css).
2. **Source de Vérité des "Design Tokens" : tailwind.config.js**
   * **Règle :** Ce fichier est notre source unique de vérité pour toutes les valeurs de design (couleurs, polices, etc.).
   * **Structure du Thème :** Le thème doit être défini dans la section theme.extend pour augmenter le thème par défaut de Tailwind plutôt que de l'écraser.
   * **Couleurs :** Définir une palette sémantique complète.

—------

// tailwind.config.js (exemple)

colors: {

primary: { DEFAULT: '#3B82F6', light: '#93C5FD', dark: '#2563EB' },

secondary: { /\* ... \*/ },

neutral: { 50: '#F9FAFB', 100: '#F3F4F6', /\* ... \*/, 900: '#111827' },

success: { /\* ... \*/ },

warning: { /\* ... \*/ },

danger: { /\* ... \*/ },

}

—-----

**Typographie :** Définir les polices de caractères du projet.

—-----

fontFamily: {

sans: ['Inter', 'ui-sans-serif', 'system-ui'],

// 'serif' si nécessaire

}

—----

**Espacements et Tailles :** Utiliser la grille d'espacement par défaut de Tailwind pour garantir la cohérence des marges, paddings et tailles. L'utilisation de valeurs arbitraires (ex: w-[123px]) est fortement déconseillée et doit être justifiée.

**Points de Rupture (Breakpoints) :** La conception doit être "Mobile First". Les breakpoints par défaut de Tailwind (sm, md, lg, xl) sont adoptés comme standard.

### **Partie 4 : Communication avec le Back-End**

Cette section définit les outils, les conventions et les processus pour toutes les interactions entre l'application Front-End et l'API Back-End.

#### **4.1. Client HTTP Centralisé (Axios)**

Pour centraliser et standardiser nos appels réseau, nous utilisons une instance unique et pré-configurée d'Axios.

1. **Technologie :** **Axios** est le client HTTP obligatoire pour le projet.
2. **Configuration d'une Instance Unique :** Une instance Axios doit être créée et configurée dans un fichier dédié (ex: /src/lib/axios.ts). Cette instance servira pour la totalité des appels API de l'application.
3. **Intercepteurs (Interceptors) :** L'instance doit être configurée avec des intercepteurs pour automatiser la gestion de l'authentification et des erreurs.

* **Exemple de configuration de l'instance Axios :** src/lib/axios.ts

—-----

import axios from 'axios';

import { useAuthStore } from '../stores/useAuthStore'; // Notre store Zustand

const apiClient = axios.create({

baseURL: import.meta.env.VITE\_API\_BASE\_URL, // Chargé depuis les variables d'environnement

headers: {

'Content-Type': 'application/json',

},

});

// --- Intercepteur de Requête ---

// S'exécute avant chaque envoi de requête

apiClient.interceptors.request.use(

(config) => {

// Récupère le token depuis le store d'authentification

const token = useAuthStore.getState().token;

if (token) {

// Si le token existe, on l'ajoute au header Authorization

config.headers.Authorization = `Bearer ${token}`;

}

return config;

},

(error) => Promise.reject(error)

);

// --- Intercepteur de Réponse ---

// S'exécute à chaque réception d'une réponse ou d'une erreur

apiClient.interceptors.response.use(

(response) => response, // Pour les réponses réussies, on ne fait rien

(error) => {

// Gestion globale des erreurs

if (error.response && error.response.status === 401) {

// Si on reçoit une erreur 401 Unauthorized (token invalide/expiré)

// On appelle l'action de déconnexion de notre store.

useAuthStore.getState().logout();

// On peut aussi rafraîchir la page pour rediriger vers le login

window.location.href = '/login';

}

return Promise.reject(error);

}

);

export default apiClient;

—------

#### **4.2. Sécurité des Types avec l'API (Type Safety)**

Pour éliminer les erreurs entre le Front-End et le Back-End, il est impératif que nos types de données soient synchronisés.

1. **Problématique :** Définir manuellement les interfaces TypeScript pour les réponses de l'API est source d'erreurs et difficile à maintenir.
2. **Solution Obligatoire : Génération Automatique de Types.**
   * **Outil :** Nous utiliserons la bibliothèque CLI **openapi-typescript**.
   * **Principe :** Cet outil lit la spécification openapi.json exposée par le Back-End et génère un fichier .ts contenant toutes les interfaces TypeScript correspondantes (schémas de réponse, corps de requête, etc.).
3. **Workflow d'Utilisation :**
   * **Étape A :** Un script doit être ajouté dans notre fichier package.json :

—----

"scripts": {

"generate-api-types": "openapi-typescript http://localhost:8000/openapi.json --output src/types/api.ts"

}

—---

* + **Étape B :** Lorsqu'une modification de l'API est effectuée par l'équipe Back-End, le développeur Front-End exécute la commande npm run generate-api-types.
  + **Étape C :** Le fichier src/types/api.ts est mis à jour automatiquement. Le compilateur TypeScript nous signalera immédiatement tout endroit de notre code qui n'est plus compatible avec la nouvelle structure de l'API.

#### **4.3. Workflow d'Authentification JWT**

Ce workflow décrit le cycle de vie du jeton d'authentification côté client.

1. **Connexion de l'Utilisateur :**
   * L'utilisateur soumet ses identifiants via le formulaire de connexion.
   * Un appel est fait à l'endpoint POST /api/v1/auth/login.
   * En cas de succès, le Back-End retourne un jeton JWT.
   * **Action Côté Client :**
     1. Le jeton JWT reçu est stocké dans le **localStorage** du navigateur pour assurer la persistance de la session entre les rafraîchissements de page.
     2. L'état global est mis à jour via useAuthStore, en stockant le token et en passant isAuthenticated à true.
2. **Persistance de la Session au Rechargement :**
   * Au chargement initial de l'application (dans App.tsx), un effet (useEffect) vérifie la présence d'un jeton dans le localStorage.
   * Si un jeton est trouvé, son contenu est décodé (sans vérifier la signature, qui est le travail du backend) pour récupérer les informations utilisateur et vérifier la date d'expiration.
   * Si le jeton n'est pas expiré, useAuthStore est hydraté avec les informations de l'utilisateur, le connectant de manière transparente.
3. **Déconnexion de l'Utilisateur :**
   * L'utilisateur clique sur le bouton de déconnexion.
   * L'action logout du store useAuthStore est appelée.
   * **Cette action DOIT effectuer les opérations suivantes :**
     1. Supprimer le jeton du localStorage.
     2. Réinitialiser l'état du store useAuthStore à ses valeurs initiales (token null, isAuthenticated: false, user: null).
   * L'utilisateur est ensuite redirigé vers la page de connexion.

### **Partie 5 : Routage et Pages**

Cette section définit la librairie, la structure et les mécanismes de protection pour la navigation au sein de notre Single-Page Application (SPA).

#### **5.1. Librairie de Routage**

1. **Technologie Obligatoire :** **React Router** en version **6 ou supérieure**.
2. **Justification :** React Router est le standard de-facto de l'écosystème React. Sa version 6+, basée sur les hooks (useNavigate, useParams, useLocation), s'intègre parfaitement à notre architecture de composants fonctionnels et offre une API déclarative puissante pour définir des mises en page et des flux de navigation complexes.

#### **5.2. Définition des Routes de l'Application**

Le tableau suivant constitue la "sitemap" de notre application pour le MVP. Il définit chaque route, le composant de page associé (situé dans /src/pages/), et le niveau d'accès requis.

| Chemin (Path) | Composant Page Associé | Accès Requis | Description |
| --- | --- | --- | --- |
| Routes Publiques |  |  |  |
| / | HomePage.tsx | Public | Page d'accueil, redirige vers /login si non authentifié, ou /dashboard si authentifié. |
| /login | LoginPage.tsx | Public | Formulaire de connexion. |
| /register | RegisterPage.tsx | Public | Formulaire d'inscription pour les apprenants et les entreprises. |
| /password-reset | PasswordResetPage.tsx | Public | Formulaire pour demander la réinitialisation du mot de passe. |
| /portfolios/:publicUrl | PublicPortfolioPage.tsx | Public | Page d'affichage d'un portfolio public. |
|  |  |  |  |
| Routes Communes (Authentifié) |  |  |  |
| /settings | SettingsPage.tsx | Authentifié | Page de gestion des paramètres du compte (profil, mot de passe). |
|  |  |  |  |
| Parcours Apprenant |  |  |  |
| /dashboard | LearnerDashboardPage.tsx | Rôle: apprenant | Tableau de bord principal de l'apprenant. |
| /projects | ProjectListPage.tsx | Rôle: apprenant | Page de recherche et de consultation de la liste des projets disponibles. |
| /projects/:projectId | ProjectDetailPage.tsx | Rôle: apprenant | Page de détail d'un projet, avec gestion des jalons et soumission des livrables. |
| /my-portfolio | MyPortfolioEditPage.tsx | Rôle: apprenant | Interface de gestion et de prévisualisation de son propre portfolio. |
|  |  |  |  |
| Parcours Entreprise |  |  |  |
| /company/dashboard | CompanyDashboardPage.tsx | Rôle: entreprise | Tableau de bord principal de l'entreprise pour suivre ses projets. |
| /company/projects/new | NewProjectPage.tsx | Rôle: entreprise | Formulaire de création d'un nouveau projet. |
| /company/projects/:projectId | CompanyProjectDetailPage.tsx | Rôle: entreprise | Page de détail d'un projet soumis, avec suivi des livrables et des évaluations. |
|  |  |  |  |
| Parcours Administrateur |  |  |  |
| /admin/dashboard | AdminDashboardPage.tsx | Rôle: admin | Tableau de bord principal de l'administrateur. |
| /admin/users | AdminUserManagementPage.tsx | Rôle: admin | Interface de gestion des utilisateurs. |
| /admin/projects | AdminProjectManagementPage.tsx | Rôle: admin | Interface de gestion et de validation des projets. |
|  |  |  |  |
| Route Erreur |  |  |  |
| \* | NotFoundPage.tsx | Public | Page affichée pour toute URL ne correspondant à aucune route définie. |

#### **5.3. Implémentation des Routes Protégées**

Pour restreindre l'accès aux pages en fonction du statut d'authentification et du rôle de l'utilisateur, nous utiliserons un composant "wrapper" réutilisable.

1. **Pattern :** Création d'un composant <ProtectedRoute />.
2. **Fonctionnement :**
   * Ce composant sera utilisé dans la configuration du routeur pour encapsuler les routes nécessitant une protection.
   * Il acceptera une prop allowedRoles (un tableau de chaînes de caractères, ex: ['admin', 'entreprise']).
   * À l'intérieur, il utilisera le hook de notre store useAuthStore pour récupérer l'état isAuthenticated et le role de l'utilisateur.
   * **Logique de Contrôle :**
     1. Si isAuthenticated est false, le composant redirigera systématiquement l'utilisateur vers la page /login via le composant <Navigate /> de React Router.
     2. Si isAuthenticated est true et que la prop allowedRoles est fournie, il vérifiera si le rôle de l'utilisateur est présent dans le tableau des rôles autorisés. Si ce n'est pas le cas, il redirigera vers une page d'erreur "Accès Interdit" (/403) ou vers le tableau de bord par défaut de l'utilisateur.
     3. Si l'utilisateur est authentifié et possède le bon rôle (ou si aucun rôle spécifique n'est requis), le composant rendra ses children (c'est-à-dire la page demandée).
3. **Exemple de Mise en Œuvre dans le Routeur Principal :**
   * **Fichier :** src/App.tsx (ou un fichier de routage dédié)

—---

import { Routes, Route } from 'react-router-dom';

import ProtectedRoute from './providers/ProtectedRoute'; // Notre composant de protection

// Import des composants de page...

import LoginPage from './pages/LoginPage';

import LearnerDashboardPage from './pages/LearnerDashboardPage';

import AdminDashboardPage from './pages/AdminDashboardPage';

function AppRoutes() {

return (

<Routes>

{/\* Route publique \*/}

<Route path="/login" element={<LoginPage />} />

{/\* Route protégée, accessible uniquement par les utilisateurs avec le rôle 'apprenant' \*/}

<Route

path="/dashboard"

element={

<ProtectedRoute allowedRoles={['apprenant']}>

<LearnerDashboardPage />

</ProtectedRoute>

}

/>

{/\* Route protégée, accessible uniquement par les utilisateurs avec le rôle 'admin' \*/}

<Route

path="/admin/dashboard"

element={

<ProtectedRoute allowedRoles={['admin']}>

<AdminDashboardPage />

</ProtectedRoute>

}

/>

{/\* ... définition des autres routes selon le même modèle \*/}

</Routes>

);

}

—--

### **Partie 6 : Qualité, Tests et Performance**

Cette section définit notre engagement envers la qualité logicielle à travers une stratégie de test rigoureuse, des conventions de code strictes et une attention constante portée à la performance.

#### **6.1. Stratégie de Test**

Les tests ne sont pas une option, mais une partie intégrante du processus de développement. Ils sont notre garantie contre les régressions et nous permettent de faire évoluer le code avec confiance.

1. **Outils de Test :**
   * **Exécuteur de Tests (Test Runner) :** **Vitest** est notre standard. Sa compatibilité native avec Vite assure une exécution quasi-instantanée des tests et une expérience de développement fluide.
   * **Bibliothèque de Test de Composants :** **React Testing Library (RTL)** est obligatoire. Sa philosophie est de tester les composants de la manière dont un utilisateur les utilise, en se concentrant sur le comportement plutôt que sur les détails d'implémentation.
2. **Typologie des Tests :**
   * **Tests Unitaires :**
     + **Objectif :** Isoler et valider la logique de fonctions pures ou de hooks personnalisés, sans aucun rendu de composant.
     + **Périmètre :** Fonctions utilitaires dans /src/lib/, hooks complexes dans /src/hooks/. Les dépendances sont mockées via les utilitaires de Vitest.
     + **Exemple Concret :** Tester un hook useDebounce pour s'assurer qu'il ne met à jour une valeur qu'après un certain délai.
   * **Tests d'Intégration de Composants :**
     + **Objectif :** C'est le cœur de notre stratégie de test. Il s'agit de tester une fonctionnalité ou une page en rendant les composants, en simulant des interactions utilisateur et en vérifiant que l'UI réagit comme prévu.
     + **Périmètre :** Composants dans /src/features/ et /src/pages/. Les appels API sont interceptés et mockés (par exemple avec msw - Mock Service Worker) pour simuler les réponses du serveur sans dépendre du réseau.
     + **Exemple Concret de Script (LoginForm.test.tsx):**

—-------

import { render, screen } from '@testing-library/react';

import userEvent from '@testing-library/user-event';

import { expect, vi } from 'vitest';

import LoginForm from './LoginForm';

// On simule (mock) la fonction de login qui serait appelée

const mockLogin = vi.fn();

test('devrait appeler la fonction de login avec les bonnes informations', async () => {

// ARRANGE : On rend le composant

render(<LoginForm onLogin={mockLogin} />);

const user = userEvent.setup();

// On récupère les champs du formulaire comme le ferait un utilisateur

const emailInput = screen.getByLabelText(/adresse e-mail/i);

const passwordInput = screen.getByLabelText(/mot de passe/i);

const submitButton = screen.getByRole('button', { name: /se connecter/i });

// ACT : On simule les interactions utilisateur

await user.type(emailInput, 'test@skillforge.ai');

await user.type(passwordInput, 'password123');

await user.click(submitButton);

// ASSERT : On vérifie que notre fonction mockée a été appelée, et avec les bons arguments.

expect(mockLogin).toHaveBeenCalledOnce();

expect(mockLogin).toHaveBeenCalledWith({

email: 'test@skillforge.ai',

password: 'password123',

});

});

—-----

#### **6.2. Qualité et Conventions de Code**

L'uniformité du code est essentielle pour la collaboration et la maintenance.

1. **Analyse Statique (Linting) : ESLint**
   * **Règle :** **ESLint** est obligatoire. Sa configuration doit inclure les plugins suivants pour garantir les meilleures pratiques : eslint-plugin-react, eslint-plugin-react-hooks, @typescript-eslint/eslint-plugin, et eslint-plugin-jsx-a11y (pour l'accessibilité).
   * **Application :** La commande de lint sera exécutée en pre-commit hook et dans le pipeline de CI.
2. **Formatage du Code : Prettier**
   * **Règle :** **Prettier** est notre formateur de code non-négociable. Il garantit un style de code unique et met fin à tout débat sur le formatage.
   * **Application :** Il sera également exécuté en pre-commit hook.

#### **6.3. Directives de Performance**

Une application moderne se doit d'être rapide. La performance est une fonctionnalité à part entière.

1. **Découpage du Code (Code Splitting) :**
   * **Technique :** Utilisation systématique de React.lazy() et du composant <Suspense />.
   * **Règle :** Tous les composants de page (dans /src/pages/) DOIVENT être importés de manière dynamique dans le fichier de configuration du routeur. Cela garantit qu'un utilisateur ne télécharge que le code JavaScript nécessaire à la page qu'il consulte.
   * **Exemple dans le routeur :**

—-----

import { lazy, Suspense } from 'react';

const LoginPage = lazy(() => import('./pages/LoginPage'));

const LearnerDashboardPage = lazy(() => import('./pages/LearnerDashboardPage'));

<Suspense fallback={<Spinner />}>

<Routes>

<Route path="/login" element={<LoginPage />} />

<Route path="/dashboard" element={<LearnerDashboardPage />} />

</Routes>

</Suspense>

—---

**Optimisation des Images :**

* **Format :** Privilégier les formats d'image modernes et performants comme **WebP** lorsque c'est possible.
* **Chargement :** Toutes les images qui ne sont pas critiques pour le premier affichage (non visibles "au-dessus de la ligne de flottaison") doivent utiliser l'attribut de chargement natif du navigateur : <img loading="lazy" ... />.

**Analyse du Bundle Final :**

* **Outil :** Utilisation d'un analyseur de bundle comme vite-plugin-visualizer.
* **Processus :** Avant chaque release majeure, l'équipe doit générer une visualisation du bundle de production.
* **Objectif :** Identifier les dépendances qui contribuent le plus à la taille finale du fichier JavaScript, s'assurer que le "tree-shaking" fonctionne correctement, et valider l'efficacité de notre stratégie de code-splitting.