

Guide Architectonique et Bonnes Pratiques : Vite + React + Tailwind CSS

Ce document consolide les standards de l'industrie et les choix architecturaux optimisés pour des projets d'envergure (comme le portail AEGIS). L'objectif est de garantir un code scalable, maintenable, sécurisé et hautement performant.

1. Architecture des Dossiers (Le modèle "Feature-based")

Pour éviter le syndrome du "plat de spaghettis" (tous les composants dans le même dossier), nous adoptons une architecture modulaire dite *Screaming Architecture*. L'arborescence doit refléter les fonctionnalités métier du projet.

```
src/
├── assets/      # Fichiers statiques non compilés (images, polices, SVG)
├── components/  # Composants UI purs, "stupides" et réutilisables globalement
│   ├── ui/     # -> Ex: Button.tsx, Input.tsx, Card.tsx (souvent générés via shadcn/ui)
│   ├── layout/ # -> Ex: Navbar.tsx, Footer.tsx, Sidebar.tsx
│   └── common/ # -> Ex: CookieBanner.tsx, ErrorBoundary.tsx
├── features/    # Logique groupée par fonctionnalité métier (Le cœur de l'app)
│   ├── auth/   # -> Tout ce qui concerne l'authentification
│   │   ├── components/ # Composants spécifiques (ex: LoginForm.tsx)
│   │   ├── hooks/     # Hooks spécifiques (ex: useLogin.ts)
│   │   └── services/   # Appels API liés à l'auth
│   └── servers/  # -> Tout ce qui concerne la gestion des serveurs
├── hooks/       # Hooks globaux réutilisables (ex: useTheme.ts, useWindowSize.ts)
├── layouts/      # Les gabarits de pages qui englobent le contenu (ex: MainLayout.tsx,
LegalLayout.tsx)
├── pages/       # Les vues principales liées au routeur (ex: GiseLandingPage.tsx). Ne
contiennent idéalement pas de logique complexe.
├── services/    # Configuration des appels API externes globaux (ex: pocketbase.ts,
axios.ts)
├── types/       # Définitions TypeScript globales (ex: user.types.ts)
├── utils/       # Fonctions utilitaires pures (ex: formatDate.ts, cn.ts)
├── App.tsx      # Point d'entrée, Configuration des Providers et du Routeur
└── index.css    # Fichier Tailwind V4 principal (Directives @theme)
```

2. Conventions de Nommage

La cohérence dans le nommage est vitale pour le repérage au sein d'une équipe.

Type de fichier / élément	Convention	Exemples
Composants React (Fichiers & Fonctions)	PascalCase	Button.tsx, HeroSection.tsx
Hooks personnalisés	camelCase avec préfixe "use"	useTheme.ts, useAuth.ts
Fonctions utilitaires & Fichiers de config	camelCase	formatDate.ts, pocketbase.ts
Interfaces / Types (TypeScript)	PascalCase (sans préfixe "I")	User, ServerConfig
Dossiers	kebab-case ou camelCase (minuscules)	components/, user-profile/

3. Bonnes Pratiques React & Design Patterns

- **Séparation des préoccupations (Custom Hooks)** : Les composants React doivent se concentrer sur l'affichage (UI). Dès qu'un composant dépasse ~150 lignes ou contient plusieurs useState / useEffect, extrayez cette logique dans un Hook personnalisé (ex: `const { data, isLoading } = useAegisData();`).
- **Le Design Pattern "Layout Wrapper"** : Ne répétez pas la Navbar et le Footer dans chaque page. Créez des composants Layout qui utilisent la prop children pour englober le contenu spécifique de la page.
- **Rendu Conditionnel Propre** : Utilisez l'opérateur logique && au lieu des ternaires nulles.
 Mauvais : `{ isLoading ? <Loader /> : null }`
 Bon : `{ isLoading && <Loader /> }`
- **Typage Strict (TypeScript)** : Le type any est formellement interdit. Tapez toujours vos props et les retours d'API.
- **Mémoisation avec useMemo et useCallback** : Utilisez useMemo pour mettre en cache des calculs coûteux et useCallback pour éviter la recreation inutile de fonctions passées en tant que props aux composants enfants, améliorant ainsi les performances.
- **Gestion Globale de l'État** : Évitez le "Prop Drilling" (passer des props à travers de multiples niveaux de composants) en utilisant l'API Context de React ou des bibliothèques de gestion d'état globales (ex: Zustand, Jotai, Redux) pour les données partagées à travers l'application.

4. Maîtriser Tailwind CSS dans React

Tailwind CSS est puissant mais peut rendre le JSX illisible. La bonne pratique absolue est l'utilisation de l'utilitaire **cn** (fusion de **clsx** et **tailwind-merge**).

L'utilitaire `src/utls/utls.ts` :

```
import { clsx, type ClassValue } from "clsx";
import { twMerge } from "tailwind-merge";

export function cn(...inputs: ClassValue[]) {
  return twMerge(clsx(inputs));
}
```

Exemple d'utilisation (pour des composants réutilisables) :

```
export function Button({ variant = 'primary', className, ...props
}: ButtonProps) {
  return (
    <button className={cn(
      "px-4 py-2 rounded-md font-bold transition-all",
      variant === 'primary' && "bg-blue-600 text-white",
      className // Permet d'écraser les classes depuis le parent
    )} {...props} />
  );
}
```

Utilisation des Plugins Tailwind : Intégrez des plugins officiels ou populaires pour étendre les capacités de Tailwind (ex: `@tailwindcss/forms` pour styliser facilement les formulaires, `@tailwindcss/typography` pour des styles de texte riches).

5. Configuration et Écosystème ViteJS

A. Les Imports Absolus (Alias)

Dites adieu aux imports relatifs illisibles comme `../../../../../components/Button`. Configurez Vite et TypeScript pour utiliser l'alias `@/` pointant vers `src/`.

- vite.config.ts : alias: { '@': path.resolve(__dirname, './src') }
- tsconfig.app.json : "paths": { "@/*": ["./src/*"] }

B. La gestion des Variables d'Environnement (.env)

Fichier	Rôle & Commit Git
.env	Variables publiques par défaut. Committé sur Git.
.env.local	Surcharges locales et clés secrètes. Strictement ignoré par Git (.gitignore).

Règles de sécurité :

1. Seules les variables préfixées par VITE_ sont injectées dans le code (ex: VITE_API_URL).
2. Accès dans le code via import.meta.env.VITE_API_URL.
3. En production (CI/CD type Gitea Actions), les secrets sont injectés à la volée avant le build via un script (ex: echo "VITE_KEY=\${{ secrets.KEY }}" >> .env.production).

C. Lazy Loading (Code Splitting)

Pour optimiser le chargement, déversez le code par "morceaux". Si un utilisateur visite la Landing Page, il ne doit pas télécharger le code du Dashboard AEGIS.

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

const GiseLandingPage = lazy(() =>
import('@pages/GiseLandingPage'));
const AegisDashboard = lazy(() =>
import('@pages/AegisDashboard'));

// Utilisation avec <Suspense fallback={<Loader/>}> autour des
Routes
```

6. Sécurité Frontend (Paradigme Cybersécurité)

- **Aucun secret lourd dans le Frontend** : Il est techniquement impossible de cacher une clé secrète dans une application Vite/React. Si vous avez besoin d'utiliser une clé secrète critique (ex: API Stripe, clé d'administration), vous devez utiliser un **Serveur Relais (BFF - Backend For Frontend)**. Le frontend interroge le backend, et le backend utilise la clé secrète pour interroger le service tiers.
- **XSS (Cross-Site Scripting)** : React échappe automatiquement les variables. N'utilisez jamais dangerouslySetInnerHTML sans avoir assaini les données avec une librairie comme *DOMPurify*.
- **Sécurité des API (PocketBase)** : Ne comptez jamais sur le fait que le frontend "cache"

des boutons pour sécuriser des actions. La sécurité doit être appliquée côté Backend avec des **API Rules strictes** (Row-Level Security) pour empêcher les requêtes non autorisées directement sur l'URL publique de l'API.

7. Composants Universels Réutilisables

Voici des exemples de composants de base que l'on retrouve dans presque toutes les applications :

A. Composant de Chargement (Loader)

```
// src/components/ui/Loader.tsx
import { Loader2 } from "lucide-react";
import { cn } from "@/utils/utils";

interface LoaderProps {
  className?: string;
  size?: number;
}

export function Loader({ className, size = 24 }: LoaderProps) {
  return (
    <Loader2
      size={size}
      className={cn("animate-spin text-blue-600
dark:text-blue-400", className)}
    />
  );
}
```

B. Composant Formulaire (Input)

```
// src/components/ui/Input.tsx
import * as React from "react";
import { cn } from "@/utils/utils";

export interface InputProps extends
React.InputHTMLAttributes<HTMLInputElement> {
  label?: string;
  error?: string;
}

export const Input = React.forwardRef<HTMLInputElement,
InputProps>(
  ({ className, type, label, error, ...props }, ref) => {
```

```

    return (
      <div className="w-full space-y-1">
        {label && (
          <label className="block text-xs font-semibold
text-slate-700 dark:text-slate-300">
            {label}
          </label>
        )}
        <input
          type={type}
          className={cn(
            "w-full bg-white dark:bg-slate-900 border
border-slate-300 dark:border-slate-800 rounded-2xl px-4 py-3
text-slate-900 dark:text-white focus:outline-none
focus:border-blue-600 focus:ring-2 focus:ring-blue-600/20 text-xs
transition-all",
            error && "border-red-500 focus:border-red-500
focus:ring-red-500/20",
            className
          )}
          ref={ref}
          {...props}
        />
        {error && <span className="text-xs
text-red-500">{error}</span>}
      </div>
    );
  }
);
Input.displayName = "Input";

```

C. Exemple de Formulaire de Connexion (Login)

```

// src/features/auth/components/LoginForm.tsx
import { useState } from 'react';
import { Button } from '@components/ui/Button';
import { Input } from '@components/ui/Input';
import { Loader } from '@components/ui/Loader';

export function LoginForm() {
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const handleSubmit = async (e: React.FormEvent<HTMLFormElement>)
=> {
    e.preventDefault();

```

```

    setIsLoading(true);
    setError(null);

    // Logique d'authentification ici...
    // Simulation
    setTimeout(() => {
        setIsLoading(false);
    }, 2000);
};

return (
    <form onSubmit={handleSubmit} className="space-y-4 w-full
max-w-sm">
        <Input
            type="email"
            label="Email"
            placeholder="votre@email.com"
            required
        />
        <Input
            type="password"
            label="Mot de passe"
            placeholder="●●●●●●●●"
            required
        />

        {error && <p className="text-red-500 text-xs">{error}</p>}

        <Button type="submit" className="w-full flex justify-center
items-center gap-2" disabled={isLoading}>
            {isLoading ? <><Loader size={16} className="text-white"/>
Connexion...</> : "Se connecter"}
        </Button>
    </form>
);
}

```