

Principes de l'architecture

Introduction

Ce document a pour vocation de définir une architecture moderne et évolutive pour l'application native Pass Culture, alignée avec notre **Vision Tribe 2025** et les besoins produit croissants.

Avec une codebase de plus de 4 ans et l'évolution rapide de l'écosystème React Native, nous devons adapter notre architecture pour répondre aux nouveaux défis : **performance < 2s**, **accessibilité RGAA**, **autonomie des squads**, et **gestion responsable des ressources**.

L'objectif est d'établir des principes consensuels et éprouvés qui guident nos décisions techniques tout en servant notre mission : **faciliter l'accès à la culture pour tous**.

Pourquoi une architecture ?

Enjeux métier Pass Culture

Une architecture bien conçue répond directement aux besoins de notre service public culturel :

- **Scalabilité utilisateurs** : Supporter la croissance et diversification des publics (15-20 ans → tous âges)
- **Fiabilité critique** : Assurer un service stable pour l'accès à la culture (crash rate <0.1%)
- **Performance inclusive** : Temps de chargement < 2s sur tous appareils et réseaux
- **Maintenance efficace** : Réduire le coût de développement pour maximiser l'investissement dans les fonctionnalités

Fondamentaux architecturaux logiciels

- **Séparation des responsabilités** : Chaque composant a un rôle unique et bien défini
- **Modularité** : Faciliter les évolutions sans impact en cascade, éventuellement aligné avec les périmètres de squads
- **Testabilité** : Permettre la validation comportementale sans couplage à l'implémentation, le plus proche du comportement de l'utilisateur

- **Prévisibilité** : Les développeurs savent intuitivement où placer et trouver le code
- **Évolutivité** : L'architecture grandit avec les besoins sans refonte majeure

Qu'est-ce qu'une bonne architecture ?

Critères de qualité pour Pass Culture

Autonomie des équipes

L'architecture doit permettre aux **3 squads** de développer en parallèle sans blocages :

- Modules découplés avec interfaces claires
- Dépendances explicites et minimales
- Standards partagés pour la cohérence

Prédictibilité du développement

Un développeur, quel que soit son niveau, doit savoir instinctivement :

- Où créer un nouveau composant selon sa responsabilité
- Comment accéder aux données (locales vs serveur)
- Quels patterns utiliser pour les cas courants

Robustesse en production

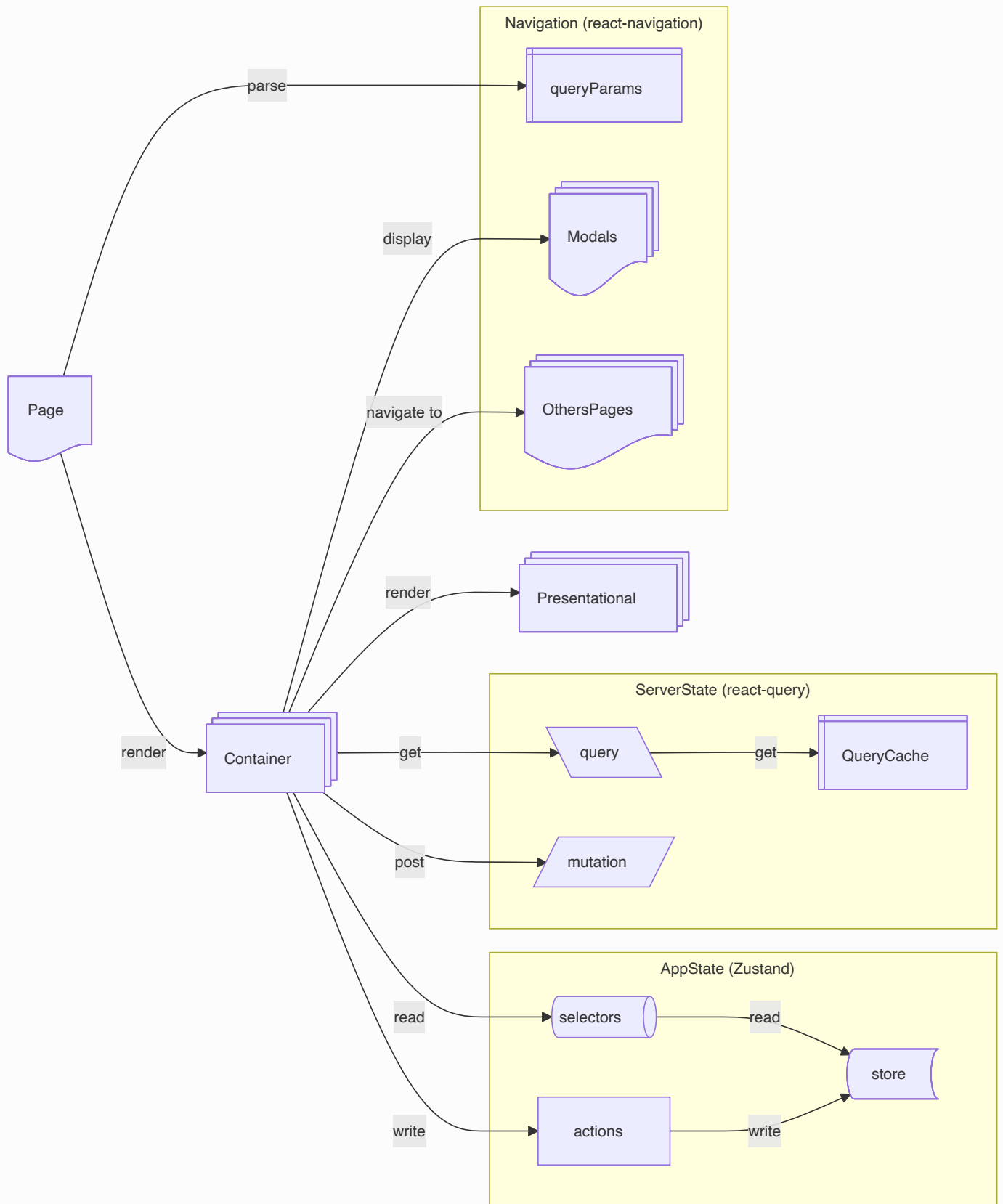
- **Environnement de test stable** : Un test simple doit toujours fonctionner
- **Contrôle des dépendances externes** : Isolation via mocks ou injection
- **Gestion d'erreur cohérente** : Comportement prévisible en cas d'échec
- **Monitoring intégré** : Visibilité sur la santé applicative (performances brutes, accessibilité, ...)

Les principes

Vue d'ensemble




Notre architecture moderne repose sur **7 principes fondamentaux** qui garantissent la **modularité**, la **séparation des responsabilités**, et l'**optimisation des performances** tout en respectant les standards actuels de React Native et surtout les contraintes du service public :

1. **Structure modulaire par fonctionnalités** avec colocation des éléments liés (feature design)
2. **Gestion d'état spécialisée** selon la nature des données (serveur/local)
3. **Séparation UI/Logique/Navigation** pour une testabilité optimale (pattern MVC)
4. **Injection de dépendances** via Context pour la flexibilité
5. **Logiques portées par le backend** pour simplifier et sécuriser
6. **Standards qualité intégrés** avec TypeScript et tests automatisés
7. **Performance et accessibilité** dès la conception



1. Structure modulaire par fonctionnalités

Pain points adressés

-  **153 fichiers >200 lignes** : la colocation réduit la charge cognitive et facilite la navigation, même si les 200 lignes sont assez faibles au regard de certains composants.
-  **97 commits context/provider** : une structure plus claire évite des modifications dispersées
-  **Maintenance burden** : des modules autonomes facilitent les évolutions et le debug
- **Responsabilité** : (continuer d') organiser le code par domaine métier plutôt que par type technique
- **Colocation** : Regrouper composants, logique, tests et styles d'une même fonctionnalité
- **Utilisation typique** : Chaque feature (search, profile, offers) est autonome et testable isolément
- **Exemple** :




```

1  src/features/search/
2    ├── pages/SearchPage.tsx
3    ├── containers/SearchContainer.tsx
4    ├── components/SearchFilters.tsx
5    ├── queries/useSearchQuery.ts
6    ├── stores/searchStore.ts
7    ├── selectors/searchSelector.ts
8    ├── fixtures/searchFixtures.ts
9    ├── constants.ts
10   ├── types.ts
11   └── __tests__/

```

2. Gestion d'état spécialisée avec React Query et Zustand

Pain points adressés

-  **28 createContext identifiés** : React Query + Zustand remplacent la prolifération de contexts
-  **22 providers actifs App.tsx** : État serveur/local séparé pour réduire drastiquement les providers
-  **Performance P95 ~4s** : Cache intelligent et re-renders optimisés pour améliorer les temps de chargement

React Query pour l'état serveur

- **Responsabilité** : Gérer cache, synchronisation, et états de chargement des données distantes

- **Colocation** : Queries dans le dossier `queries/` de chaque feature
- **Utilisation typique** : API calls, cache automatique, revalidation en arrière-plan

Sélecteurs pour éviter les re-renders :

Un sélecteur permet d'encapsuler la logique de retrait d'une valeur spécifique d'un state (une dérivée) pour améliorer les performances.

```
1 // ✅ Bon pattern avec sélecteur
2 const selectArtistsNb = (artists: Artists) => artists.length
3
4 const ArtistContainer = () => {
5   const { data: artistsNb } = useArtistsQuery({ select: selectArtistsNb })
6   return <Text>{artistsNb}</Text>
7 }
```

Contrat de query complet :

```
1 // ✅ Retourner l'intégralité de la query
2 const useArtistsQuery = () => useQuery({ queryFn: fetchArtists, queryKey: ['artists'] })
```

Gestion d'état déclarative avec Suspense :

<https://react.dev/reference/react/Suspense>

<https://react.dev/reference/react/Component#catching-rendering-errors-with-an-error-boundary>

```
1 // Au niveau navigation - erreurs dures
2 <ErrorBoundary fallback={<PageNotFound />}>
3   <Suspense fallback={<LoadingPage />}>
4     <SomePage />
5   </Suspense>
6 </ErrorBoundary>
7
8 // Au niveau container - erreurs souples
9 <ErrorBoundary fallback={null}>
10   <Suspense fallback={<LoadingContainer />}>
11     <OptionalContainer />
12   </Suspense>
13 </ErrorBoundary>
```

Zustand pour l'état local




- **Responsabilité** : Gérer l'état applicatif local (UI, préférences, navigation)
- **Colocation** : Stores dans `stores/` par domaine métier

- **Utilisation typique** : Éviter le prop drilling et les Context pour l'état

```
1 interface UserState {
2   preferences: UserPreferences
3   setPreferences: (prefs: UserPreferences) => void
4 }
5
6 export const useUserStore = create<UserState>((set) => ({
7   preferences: defaultPreferences,
8   setPreferences: (preferences) => set({ preferences }),
9 })))
```

3. Séparation des responsabilités : Pages, Containers, Components

Pain points adressés

-  **Complexité cognitive SonarCloud de 58** : Séparation claire des responsabilités pour réduire la complexité
-  **Test environment instable** : Composants découplés de la logique pour faciliter les tests isolés
-  **153 fichiers >200 lignes** : Composants plus petits et focalisés

Pages : Navigation et paramètres

- **Responsabilité** : Accéder aux paramètres de navigation et orchestrer les containers
- **Colocation** : Dans `src/features/<feature>/pages/`
- **Utilisation typique** : Point d'entrée d'un écran avec ses paramètres de route

```
1 const ArtistsPage: FunctionComponent = () => {
2   const route = useRoute<UseRouteType<'Artists'>>()
3   return (
4     <ArtistContainer artistId={route.params.artistId} />
5   )
6 }
```

Containers : Données et logique métier

- **Responsabilité** : Gérer les données pour les composants enfants : consommer les hooks, gérer l'état et en définitive orchestrer les composants visuels

- **Colocation** : Dans `src/features/<feature>/containers/`
- **Utilisation typique** : Interface entre les données et l'affichage

```

1  type Props = { artistId: string }
2
3  const ArtistsContainer: FunctionComponent<Props> = ({ artistId }) => {
4    const { data: artist } = useArtistQuery(artistId)
5    const onPress = () => analytics.logConsultOffer()
6
7    return (
8      <ArtistCard
9        name={artist.name}
10       imageURL={artist.image}
11       onPress={onPress}
12     />
13   )
14 }

```

Components : UI pure et réutilisable

- **Responsabilité** : Affichage uniquement, sans logique métier ni effets de bord
- **Colocation** : Dans `src/features/<feature>/components/` ou `src/shared/ui/`
- **Utilisation typique** : Composants réutilisables testables en isolation

```

1  type Props = {
2    name: string
3    imageURL: string
4    onPress: () => void
5  }
6
7  const ArtistCard: FunctionComponent<Props> = ({ name, imageURL, onPress }) => {
8    return (
9      <Pressable onPress={onPress}>
10        <Image source={{ uri: imageURL }} />
11        <Text>{name}</Text>
12      </Pressable>
13    )
14 }

```

Principe anti-couplage API :

```




1  // ❌ Props couplées à l'API
2  type Props = { artist: ArtistResponse | null }
3
4  // ✅ Props découplées
5  type Props = { artistName: string; artistImageURL: string }



```


4. Context API pour l'injection de dépendances

L'injection de dépendance facilite la testabilité et l'évolutivité du système.

Pain points adressés

-  **28 contextes = maintenance nightmare** : Context réservé aux services (les services sont les dépendances à injecter), pas à l'état
-  **Test environment instable** : Injection pour faciliter les mocks et les tests
-  **Éviter re-renders Context** : Plus d'état dans Context = plus de problèmes performance via les re-renders
- **Responsabilité** : Fournir des services et des abstractions, et ne pas gérer l'état
- **Colocation** : Dans `src/shared/providers/` pour les services transversaux
- **Utilisation typique** : Injection de services, configuration, thème

```
1 // 
2 const UserServiceContext = createContext<UserService | null>(null)
3
4 export const UserProvider = ({ children }: { children: ReactNode }) => {
5   const userService = useMemo(() => new UserService(), [])
6   return (
7     <UserServiceContext.Provider value={userService}>
8       {children}
9     </UserServiceContext.Provider>
10  )
11 }
12
13 //  Anti-pattern : Context comme store global
14 const UserStateContext = createContext({
15   user: null,
16   setUser: () => {},
17   isLoading: false,
18   setIsLoading: () => {},
19   error: null,
20   setError: () => {},
21   preferences: {},
22   setPreferences: () => {}
23 })
24
25 export const UserProvider = ({ children }) => {
26   const [user, setUser] = useState(null)
27   const [isLoading, setIsLoading] = useState(false)
```




```

28   const [error, setError] = useState(null)
29   const [preferences, setPreferences] = useState({})
30
31   // Chaque changement re-render tous les enfants
32   return (
33     <UserStateContext.Provider
34       value={{ user, setUser, isLoading, setIsLoading, error, setError, preferences,
35       setPreferences }}
36     >
37       {children}
38     </UserStateContext.Provider>
39   )
40 }
41 // ❌ Anti-pattern : Service recréé sans memoization
42 export const ApiProvider = ({ children }) => {
43   // Nouvelle instance à chaque render = re-render cascade
44   const apiService = new ApiService()
45   const analyticsService = new AnalyticsService()
46
47   return (
48     <ApiContext.Provider value={{ apiService, analyticsService }}>
49       {children}
50     </ApiContext.Provider>
51   )
52 }

```

5. Logiques portées par le backend

Pain points adressés

-  **Complexité frontend excessive** : Calculs métier côté serveur pour réduire la complexité côté mobile et l'unicité des règles métier, voire le partage avec Pro...
-  **Performance P95 ~4s** : Données pré-formatées pour réduire les temps de traitement
-  **Bundle Android 18.7MB/iOS 33.8MB** : Moins de logique = moins de code = bundle plus petit
- **Responsabilité** : Déplacer la complexité métier côté serveur pour simplifier le frontend react
- **Colocation** : API endpoints dédiés avec données pré-formatées pour l'affichage
- **Utilisation typique** : Calculs complexes, règles métier, formatage données

Avantages architecturaux :

- **Testabilité améliorée** : Logique métier peut être testée sans couplage avec l'UI

- **Performance optimisée** : Calculs côté serveur, cache de données pré-calculées, données optimisées pour leur consommation côté mobile
- **Cohérence garantie** : Une seule source de vérité pour les règles métier
- **Évolutivité** : Modifications de règles métier sans redéploiement mobile

```

1 // ❌ Calcul côté frontend
2 const useOfferPriority = (user: User, offers: Offer[]) => {
3   return useMemo(() => {
4     return offers.map(offer => ({
5       ...offer,
6       priority: calculatePriority(user.age, user.preferences, offer.category)
7     }))
8   }, [user, offers])
9 }
10
11 // ✅ Données pré-calculées par le backend
12 const usePersonalizedOffers = (userId: string) => {
13   return useQuery({
14     queryKey: ['offers', 'personalized', userId],
15     queryFn: () => fetchPersonalizedOffers(userId) // API retourne offers avec priority
16   })
17 }

```

6. Tests unitaires et isolation du système à tester

Pain points adressés

- **✅ Test environment instable** : Isolation des dépendances pour garantir des tests fiables
- **✅ 97 commits context/provider** : Tests comportementaux qui résistent aux refactors
- **✅ Complexité maintenance** : Tests focalisés sur comportements vs implémentation (trophée de test)
- **Responsabilité** : Tester les comportements sans couplage à l'implémentation (mais à l'utilisateur)
- **Colocation** : Tests à côté du code testé dans chaque feature
- **Utilisation typique** : Isolation des dépendances externes, tests de comportement

Test environment stable :

```

1 // ✅ Ce test doit toujours fonctionner
2 render(<ComplexComponent />)
3 expect(true).toBeTruthy()

```

```

1 // ✅ Bon pattern : Test du comportement utilisateur
2 const ArtistContainer = () => {
3   const { data: artist, isLoading } = useArtistQuery()
4
5   if (isLoading) return <Loading />
6   return <ArtistCard name={artist.name} />
7 }
8
9 // Test focalisé sur l'expérience utilisateur
10 test('should show loading then artist name', async () => {
11   // Setup: API retourne des données réelles via MSW
12   server.use(
13     rest.get('/api/artists/123', (req, res, ctx) =>
14       res(ctx.json({ name: 'Real Artist' })))
15   )
16 )
17
18 render(<ArtistContainer artistId="123" />)
19
20 // Comportement: utilisateur voit loading puis contenu
21 expect(screen.getByText('Chargement...')).toBeInTheDocument()
22 await waitFor(() => {
23   expect(screen.getByText('Real Artist')).toBeInTheDocument()
24 })
25 })
26
27 // ✅ Bon pattern : Mock les services externes, pas nos hooks
28 beforeEach(() => {
29   // Mock la librairie externe, pas notre logique
30   jest.mock('algoliasearch', () => ({
31     search: jest.fn().mockResolvedValue({
32       hits: [{ name: 'Artist from Algolia' }]
33     })
34   }))
35 })
36
37 test('should display search results from API', async () => {
38   render(<SearchContainer query="artist" />)
39
40   // Notre useSearchQuery utilise la vraie logique avec Algolia mocké
41   await waitFor(() => {
42     expect(screen.getByText('Artist from Algolia')).toBeInTheDocument()
43   })
44 })

```

Règles d'isolation :

- Injecter un service de test (ou mocker) les librairies externes (`algoliasearch`), pas nos hooks
- Mocker le backend via MSW, pas les appels individuels

- Tester les comportements utilisateur, pas l'implémentation

Anti-patterns à éviter pour les Tests




```
1 // ❌ Anti-pattern : Test fragile couplé aux détails internes
2 const ArtistContainer = () => {
3   const [loading, setLoading] = useState(false)
4   const [data, setData] = useState(null)
5
6   useEffect(() => {
7     setLoading(true)
8     fetchArtist().then(setData).finally(() => setLoading(false))
9   }, [])
10
11   return loading ? <Loading /> : <ArtistCard data={data} />
12 }
13
14 // Test cassant à chaque refactor
15 test('should set loading to true then false', () => {
16   const { rerender } = render(<ArtistContainer />)
17   expect(mockSetLoading).toHaveBeenCalledTimes(1)
18   expect(mockSetLoading).toHaveBeenCalledWith(true)
19   expect(mockSetLoading).toHaveBeenCalledWith(false)
20 })
21
22 // ❌ Anti-pattern : Mock nos propres hooks = masque les régressions
23 test('should display artist name', () => {
24   jest.mock('./useArtistQuery', () => ({
25     useArtistQuery: () => ({ data: { name: 'Fake Artist' } })
26   }))
27
28   render(<ArtistContainer />)
29   expect(screen.getByText('Fake Artist')).toBeInTheDocument()
30 })
31 // Si useArtistQuery casse, le test passe toujours
```

Problèmes générés :

- Tests cassent à chaque refactor (couplage/dépendance à l'implémentation)
- Fausse confiance (mocks cachent les vrais bugs)
- Maintenance test = 2x temps développement feature
- Environment instable = CI/CD non fiable

7. Performance et accessibilité intégrées


Pain points adressés

-  **Bundle Android 18.7MB → objectif <15MB** : Architecture optimisée pour réduire le bundle
-  **Performance P95 ~4s → objectif <2s** : Patterns optimisés pour améliorer le temps de chargement et les performances d'affichages générales
-  **Accessibilité manquante** : Standards RGAA intégrés par le design, elle est intégrée dès la conception de l'architecture, pas ajoutée après coup
- **Responsabilité** : Respecter les objectifs Vision 2025 par notre design architectural
- **Colocation** : Standards appliqués dans chaque composant et feature
- **Utilisation typique** : RGAA AA, performance <2s, bundle <15MB android

Composition > props drilling :

```
1 //  Composition simple et performance
2 const Container: FunctionComponent<{ subtitle: string }> = ({ subtitle }) => {
3   const onPress = useOnPress()
4   const title = useGetTitle()
5
6   return (
7     <View>
8       <Title>{title}</Title>
9       {subtitle && <Text>{subtitle}</Text>}
10      <Button title="action" onPress={onPress} />
11    </View>
12  )
13 }
```

Anti-patterns à éviter pour Performance et Accessibilité

```
1 //  Anti-pattern : Props drilling + re-renders excessifs
2 const Container: FunctionComponent = () => {
3   const [user, setUser] = useState()
4   const [theme, setTheme] = useState()
5   const [analytics, setAnalytics] = useState()
6
7   return (
8     <ArtistSection
9       user={user}
10      theme={theme}
11      analytics={analytics}
12      onUserUpdate={setUser}
13      onThemeChange={setTheme}
14    />
```

```

15   )
16 }
17
18 const ArtistSection = ({ user, theme, analytics, onUserUpdate, onThemeChange }) => {
19   return (
20     <ArtistCard
21       user={user}
22       theme={theme}
23       analytics={analytics}
24       onUserUpdate={onUserUpdate}
25       onThemeChange={onThemeChange}
26     />
27   )
28 }
29
30 // Chaque changement user/theme re-render toute la hiérarchie
31
32 // ❌ Anti-pattern : Accessibilité comme afterthought
33 const OfferCard = ({ title, price }) => {
34   return (
35     <Pressable onPress={handlePress}>
36       <Image source={offerImage} />
37       <Text>{title}</Text>
38       <Text>{price}</Text>
39       {/* Pas de labels, pas de roles, pas de hints */}
40     </Pressable>
41   )
42 }

```

Problèmes générés :

- Performance dégradée par re-renders inutiles
- Bundle alourdi par props drilling
- Accessibilité impossible à rattraper sans refactor complet
- Maintenance complexifiée par couplages

Standards techniques et outils

TypeScript strict

- Interfaces pour tous les props et types métier
- Pas de `any`, utilisation de types utilitaires

- Sécurité de type pour les appels API

Conventions de nommage

- Queries : `useArtistsQuery.ts`
- Mutations : `useArtistsMutation.ts`
- Stores : `useArtistStore.ts`
- Components : `ArtistCard.tsx`

Outils de qualité

- ESLint avec des règles strictes
- Tests avec React Native Testing Library
- Performance monitoring intégré, comparés, analysés
- Bundle analyzer pour optimisation
- Sonar pour analyser et suivre la qualité du code