extranuts 20250615 2353 plan de protection gemini

Stratégie Architecturale pour une Application Tauri/SolidJS Évolutive et Résiliente

Section 1: Le Schéma Directeur Architectural : Un Modèle "Feature-Store"

Cette section établit la recommandation architecturale fondamentale en disséquant les options disponibles et en justifiant le modèle "Feature-Store" proposé comme le choix optimal pour le contexte spécifique de ce projet.

1.1. Déconstruction des Options : Composants, Hooks et Stores

L'architecture actuelle, un monolithe au sein de `App.tsx`, présente des risques significatifs de régression et des défis de maintenance à mesure que l'application se complexifie. Pour évoluer vers une structure plus robuste, il est essentiel d'analyser les différents patrons de conception disponibles dans l'écosystème SolidJS. Trois approches principales se dégagent : la gestion d'état par composants autonomes, l'utilisation de hooks personnalisés, et la mise en place de services ou de stores.

1.1.1. Analyse de l'Option A : Composants Autonomes avec Interfaces Strictes

La première approche consiste à s'appuyer sur la nature componentielle de SolidJS, où chaque composant gère son propre état et communique avec ses enfants via des `props`. SolidJS excelle dans la création de composants fonctionnels, réactifs et réutilisables, qui encapsulent la logique de l'interface utilisateur et l'état local.1 Cette méthode est

parfaitement adaptée pour des éléments d'interface simples et isolés, comme un bouton stylisé ou une carte d'information statique.

Cependant, pour des fonctionnalités complexes et interconnectées telles que la barre de recherche, le modal d'export ou le système de catégories, cette approche révèle rapidement ses limites. La nécessité de partager un état ou des actions entre des composants distants dans l'arborescence conduit inévitablement au "prop drilling". Ce phénomène consiste à faire transiter des `props` à travers de multiples niveaux de composants intermédiaires qui n'en ont pas directement l'utilité. Le "prop drilling" rend le flux de données difficile à suivre, complexifie le refactoring et augmente la fragilité du code. Toute modification de la signature des données partagées nécessite des ajustements dans toute la chaîne de `props`, ce qui est une source fréquente d'erreurs et de régressions.3 Dans le contexte d'une application où des fonctionnalités comme la recherche doivent interagir avec la liste principale des notes, s'appuyer uniquement sur les

`props` pour la communication d'état est une stratégie insoutenable à long terme.

1.1.2. Analyse de l'Option B : Hooks Personnalisés

L'idée d'utiliser des hooks personnalisés, comme `useSearch` ou `useExport`, est séduisante car elle semble promettre une logique réutilisable et encapsulée. Cependant, il est crucial de comprendre une différence fondamentale entre les hooks de SolidJS et ceux de React. Dans SolidJS, les "hooks" sont en réalité des fonctions de composition qui assemblent des primitives réactives (`createSignal`, `createEffect`, etc.).4 Contrairement aux hooks de React, ils ne sont pas intrinsèquement liés à un cycle de vie d'état propre à une instance de composant.

En conséquence, chaque appel à un hook personnalisé dans un composant différent crée une *nouvelle instance indépendante* des primitives réactives qu'il contient.6 Par exemple, si un composant

`SearchBar` et un composant `NoteList` appelaient tous deux un hook `useSearch`, ils obtiendraient chacun leur propre état `searchQuery`. L'état ne serait pas partagé entre eux. Ce comportement rend les hooks personnalisés inadaptés à la gestion d'un *état partagé* ou global. Ils restent cependant un excellent outil pour encapsuler une logique réutilisable mais *sans état* (stateless), ou une logique dont l'état est

destiné à rester local à l'instance du composant qui l'appelle. Pour des fonctionnalités globales comme la recherche, qui doivent affecter plusieurs parties de l'interface, les hooks personnalisés seuls ne sont pas la solution adéquate.

1.1.3. Analyse de l'Option C : Services/Stores Séparés

Cette troisième option, qui s'appuie sur des stores, est la plus prometteuse pour répondre aux exigences du projet. SolidJS fournit une primitive `createStore` spécifiquement conçue pour gérer des structures de données complexes, imbriquées et partagées.3 Un store permet de centraliser l'état et les actions qui le modifient, offrant une source unique de vérité.

Cependant, l'implémentation d'un unique store global pour toute l'application, bien que simple à démarrer, recrée un monolithe au niveau de la logique. Un tel "god store" deviendrait rapidement un goulot d'étranglement, un point de défaillance unique, et violerait le principe de séparation des préoccupations. Toutes les fonctionnalités de l'application dépendraient d'une seule et même structure de données, rendant les modifications risquées et la compréhension du code difficile. Cette approche ne résout pas le problème de modularité à la racine ; elle ne fait que déplacer le monolithe de `App.tsx` vers un fichier `globalStore.ts`. La véritable solution réside dans une application plus granulaire du concept de store.

1.2. Modèle Recommandé : L'Architecture Hybride "Feature-Store"

Face aux limites des options précédentes, la recommandation est d'adopter une architecture hybride, structurée et modulaire, que l'on peut nommer le modèle "Feature-Store". Ce modèle organise l'application non pas par couches techniques (un dossier pour tous les composants, un autre pour tous les stores), mais par "fonctionnalités" (features). Cette approche est une bonne pratique reconnue pour améliorer la modularité, la maintenabilité et l'évolutivité des applications.8

Les principes fondamentaux de ce modèle sont les suivants :

- **La Fonctionnalité comme Unité de Modularité :** Chaque fonctionnalité distincte de l'application (Recherche, Export, Catégories,

WikiLinks) est encapsulée dans son propre module, physiquement situé dans un répertoire dédié, par exemple `src/features/`. Cette co-location physique de tout le code lié à une fonctionnalité (état, logique, UI, tests) crée une encapsulation logique forte.

- **Le Store comme Cerveau de la Fonctionnalité :** Au cœur de chaque module de fonctionnalité se trouve un `createStore` dédié. Ce store est responsable de l'ensemble de l'état de la fonctionnalité, de ses états dérivés (calculés via `createMemo` pour l'efficacité), et de toute la logique métier associée (les actions qui modifient l'état).3 Cette approche est en parfaite adéquation avec la philosophie de "Données Déclaratives" de SolidJS, où le comportement d'une donnée est lié à sa déclaration.10
- **Le Contexte comme Mécanisme de Livraison :** L'instance du store de la fonctionnalité est mise à disposition des composants qui en ont besoin via les primitives `createContext` et `useContext` de SolidJS. Ce mécanisme découple proprement la logique d'état de l'interface utilisateur, élimine le "prop drilling" et permet une injection de dépendances claire et explicite.3
- **Les Composants comme Vue :** Les composants au sein du module de fonctionnalité, ainsi que ceux qui en dépendent à travers l'application, deviennent des entités "légères". Leur rôle principal est de rendre l'interface utilisateur en se basant sur l'état fourni par le store (consommé via `useContext`) et de déclencher des actions sur ce même store en réponse aux interactions de l'utilisateur.

Pour un développeur travaillant seul, ce modèle apporte une discipline et une structure essentielles. Il facilite la navigation dans le code, l'isolation des bogues et le développement de nouvelles fonctionnalités sans craindre de provoquer des défaillances en cascade. Plus important encore, il soutient directement l'évolution indépendante des fonctionnalités, une exigence clé du projet.

1.3. Matrice de Comparaison des Modèles Architecturaux

Pour synthétiser l'analyse et justifier la recommandation, la matrice suivante évalue chaque modèle architectural par rapport aux objectifs critiques du projet.

Modèle Architectural	Complexité de la Gestion d'État	Évolutivité & Modularité	Testabilité	In le Pe
Composants Autonomes (Props)	Faible (pour état local) à Élevée (pour état partagé, "prop drilling")	Faible. Fort couplage entre les composants parents et enfants. Le refactoring est fragile.	Moyenne. Les composants peuvent être testés isolément, mais tester les flux de données complexes est difficile.	Fa Co mo Sc ma "p dr pe er de re ir ma
Hooks Personnalisés (Logique seule)	Non applicable pour l'état partagé. Ne résout pas le problème de la centralisation de l'état global.	Faible. Excellent pour la logique réutilisable, mais ne fournit pas de structure pour l'état partagé.	Élevée. La logique pure dans les hooks est facile à tester unitairement.	Fa ho de fo tr pe
Store Global Monolithique	Moyenne. Centralise la logique, mais crée une dépendance globale et un couplage fort de toutes les fonctionnalités.	Faible. Le store devient un nouveau monolithe. Les modifications sont risquées et impactent toute l'application.	Faible. Tester une petite partie de la logique nécessite de mocker un store potentiellement énorme.	Mo Pe er de jo ir le de ne fi gé

Modèle Architectural	Complexité de la Gestion d'État	Évolutivité & Modularité	Testabilité	In le Pe
Modèle Feature-Store (Recommandé)	Faible à Moyenne. La complexité est contenue au sein de chaque fonctionnalité, ce qui la rend gérable.	**Élevée.** Chaque fonctionnalité est un module indépendant, découplé des autres. Évolution et refactoring sécurisés.	**Élevée.** Les stores peuvent être testés en isolation (logique métier), et les composants sont testés pour leur intégration avec le store.	Él L gi et m se st op le dé ac le pr pe de

Cette analyse comparative démontre que le modèle "Feature-Store" est le seul à répondre de manière satisfaisante à l'ensemble des contraintes et des objectifs du projet : modularité, protection contre les régressions, performance et maintenabilité à long terme, en particulier dans un contexte de développement solo où la clarté et la structure sont primordiales.

Section 2: Implémentation Pratique : Architecturer la Fonctionnalité de Recherche en Temps Réel

Pour rendre le modèle architectural "Feature-Store" concret et fournir un modèle directement applicable, cette section détaille son implémentation complète en utilisant la fonctionnalité de barre de recherche comme exemple.

2.1. Structure du Répertoire du Module de Fonctionnalité

L'adoption d'une structure de répertoires standardisée et cohérente pour chaque fonctionnalité est la première étape vers une base de code propre et maintenable. Elle favorise l'encapsulation et la découvrabilité.8

La structure recommandée pour le module de recherche est la suivante :

```
src/
└── features/
    └── search/
        ├── components/
        │   └── SearchBar.tsx
        ├── tests/
        │   ├── search.store.test.ts
        │   └── SearchBar.test.tsx
        ├── SearchProvider.tsx
        ├── search.store.ts
        └── index.ts
```

Rôle de chaque fichier :

- `search.store.ts` : Le cœur logique de la fonctionnalité. Il contient la définition du store avec `createStore`, les états dérivés avec `createMemo`, et toutes les actions de modification de l'état.
- `components/` : Un sous-répertoire pour tous les composants Reactifs (SolidJS) liés à cette fonctionnalité. Dans ce cas, `SearchBar.tsx`.
- `SearchProvider.tsx` : Un composant dédié à la création du contexte (`createContext`) et à la fourniture de l'instance du store à l'arbre des composants.
- `tests/` : Un répertoire qui co-localise tous les tests relatifs à la fonctionnalité. Cela inclut les tests unitaires pour le store et les tests d'intégration pour les composants.
- `index.ts` : Le point d'entrée public du module. Il exporte les éléments qui doivent être accessibles depuis l'extérieur du module (par exemple, le `SearchProvider` et le composant `SearchBar`), masquant ainsi les détails d'implémentation internes.

2.2. Le Store de la Fonctionnalité (`search.store.ts`)

Ce fichier définit l'état et le comportement de la fonctionnalité de recherche. Il est totalement découplé de l'interface utilisateur.

TypeScript

```
// src/features/search/search.store.ts

import { createStore, produce } from "solid-js/store";
import { createMemo } from "solid-js";
```

```
// 1. Définir les types pour l'état et les notes
export interface Note {
  id: string;
  title: string;
  content: string;
  tags: string;
}

export interface SearchState {
  readonly query: string;
  readonly allNotes: Note;
  readonly isLoading: boolean;
}

export interface SearchActions {
  setQuery: (query: string) => void;
  loadNotes: (notes: Note) => void;
  clearSearch: () => void;
}

export interface SearchStore {
  state: SearchState;
  actions: SearchActions;
  filteredNotes: () => Note;
}

// 2. La fonction de création du store
export function createSearchStore(initialNotes: Note =): SearchStore {
  const = createStore<SearchState>({
    query: "",
    allNotes: initialNotes,
    isLoading: false,
  });

  // 3. Définir les états dérivés avec createMemo pour la performance
  const filteredNotes = createMemo(() => {
    const q = state.query.toLowerCase().trim();
    if (!q) {
      return state.allNotes;
    }
    return state.allNotes.filter(
      (note) =>
        note.title.toLowerCase().includes(q) ||
```

```
        note.content.toLowerCase().includes(q)
      );
    });

    // 4. Définir les actions qui modifient le store
    const actions: SearchActions = {
      setQuery(query: string) {
        setState("query", query);
      },
      loadNotes(notes: Note) {
        // Utilise produce pour des mises à jour complexes et sûres
        setState(
          produce((s) => {
            s.allNotes = notes;
          })
        );
      },
      clearSearch() {
        setState("query", "");
      },
    };

    return { state, actions, filteredNotes };
}
```

Points clés de cette implémentation :

- **Typage Strict :** L'utilisation d'interfaces TypeScript (`SearchState`, `SearchActions`, `SearchStore`) garantit la sécurité de type et sert de documentation auto-validée pour le contrat du store.12
- **État Dérivé Efficace :** `createMemo` est utilisé pour calculer `filteredNotes`. C'est une optimisation cruciale. La logique de filtrage, potentiellement coûteuse, ne s'exécutera que lorsque `state.query` ou `state.allNotes` changeront, et non à chaque rendu ou accès à la valeur.13
- **Séparation Commande/Requête (CQRS) :** L'état (`state`) est en lecture seule de l'extérieur, et les modifications ne peuvent se faire qu'à travers les `actions` exportées. Cela crée un flux de données unidirectionnel et prévisible, une pratique encouragée par SolidJS.15

2.3. Le Fournisseur de Contexte (`SearchProvider.tsx`)

Ce composant sert de pont entre la logique du store et l'arbre des composants de SolidJS. Il instancie le store et le rend disponible via le contexte.

TypeScript

```
// src/features/search/SearchProvider.tsx

import { createContext, useContext, ParentComponent } from "solid-js";
import { createSearchStore, SearchStore, Note } from "./search.store";

// 1. Créer le contexte avec une valeur par défaut (peut être undefined)
const SearchContext = createContext<SearchStore>();

// 2. Définir le composant Provider
interface SearchProviderProps {
  initialNotes?: Note;
}

export const SearchProvider: ParentComponent<SearchProviderProps> = (props) =>
{
  // Instancier le store une seule fois
  const store = createSearchStore(props.initialNotes);

  return (
    <SearchContext.Provider value={store}>
      {props.children}
    </SearchContext.Provider>
  );
};

// 3. Créer un hook personnalisé pour consommer le contexte facilement
export function useSearch() {
  const context = useContext(SearchContext);
  if (!context) {
    throw new Error("useSearch must be used within a SearchProvider");
  }
  return context;
}
```

Ce fichier met en œuvre le patron de conception standard pour le partage d'état avec le contexte dans SolidJS.3 Le hook

`useSearch` simplifie la consommation du contexte et fournit une erreur claire si un composant tente de l'utiliser en dehors du `Provider`,

prévenant ainsi les bogues d'exécution.

2.4. Le Composant de Vue (`SearchBar.tsx`)

Le composant `SearchBar` est maintenant un composant "idiot" (dumb component). Il ne contient aucune logique d'état ; son seul rôle est d'afficher l'interface et de déléguer les interactions de l'utilisateur au store via le hook `useSearch`.

TypeScript

```
// src/features/search/components/SearchBar.tsx

import type { Component } from "solid-js";
import { useSearch } from "../SearchProvider";

export const SearchBar: Component = () => {
  // 1. Accéder au store via le hook de contexte
  const { state, actions } = useSearch();

  const handleInput = (e: Event) => {
    const target = e.currentTarget as HTMLInputElement;
    actions.setQuery(target.value);
  };

  return (
    <div>
      <input
        type="search"
        placeholder="Rechercher des notes..."
        value={state.query}
        onInput={handleInput}
        aria-label="Barre de recherche"
      />
    </div>
  );
};
```

Cette refonte illustre parfaitement la séparation des préoccupations. Le composant est simple, déclaratif et facile à tester visuellement ou avec des outils comme Storybook. Toute la complexité est gérée dans le store.

2.5. Intégration dans l'Application (`App.tsx`)

Enfin, pour activer la fonctionnalité, il suffit d'envelopper les parties de l'application qui ont besoin d'accéder à la recherche (la barre de recherche elle-même, la liste des notes, etc.) avec le `SearchProvider`.

TypeScript

```
// src/App.tsx (extrait simplifié)

import { Component } from "solid-js";
import { SearchProvider } from "./features/search"; // Importe depuis le point
d'entrée du module
import { SearchBar } from "./features/search";
import { NoteList } from "./components/NoteList"; // Un composant qui
utilisera aussi useSearch()

const App: Component = () => {
  const initialNotes =;

  return (
    <SearchProvider initialNotes={initialNotes}>
      <main>
        <header>
          <SearchBar />
        </header>
        <section>
          <NoteList />
        </section>
      </main>
    </SearchProvider>
  );
};

export default App;
```

L'importation depuis `./features/search` utilise le fichier `index.ts` du module, qui expose uniquement les parties publiques (`SearchProvider`, `SearchBar`), respectant ainsi le principe d'encapsulation.

Cette approche structurée transforme une architecture monolithique et fragile en un système modulaire, prévisible et performant. Le fait de co-localiser tous les artefacts d'une fonctionnalité (logique, UI,

tests, types) dans un seul répertoire réduit considérablement la charge cognitive. Lorsqu'une modification est nécessaire sur la recherche, le développeur sait exactement où regarder : tout se trouve dans `src/features/search/`. C'est un avantage fondamental par rapport à la navigation entre des répertoires `components`, `stores`, et `hooks` de haut niveau, où les liens entre les fichiers sont implicites et difficiles à suivre.

Section 3: Une Défense Multi-Couches Contre la Régression

Pour sécuriser les fonctionnalités critiques contre les régressions, il est inefficace de s'appuyer sur une seule méthode. Les différentes options de protection (tests, documentation, versioning) ne sont pas mutuellement exclusives ; au contraire, elles sont complémentaires. La stratégie la plus robuste consiste à les combiner dans un modèle de "défense en profondeur" (`defense-in-depth`), où chaque couche offre une protection contre les types d'erreurs que les autres pourraient laisser passer.

3.1. Couche 1 : Tests Automatisés (Le Contrat Renforcé par la Machine)

Les tests automatisés constituent la première ligne de défense. Ils fournissent un retour d'information rapide et fiable à chaque modification du code. La philosophie de test doit se concentrer sur la validation du comportement de l'application du point de vue de l'utilisateur, plutôt que sur les détails d'implémentation, une approche promue par la famille d'outils Testing Library.16

3.1.1. Tester le Store (`search.store.test.ts`)

Le store, contenant la logique métier pure, doit être testé de manière unitaire. Ces tests s'exécutent rapidement et valident que le cœur de la fonctionnalité se comporte comme prévu, indépendamment de toute interface utilisateur. Le framework de test recommandé pour SolidJS est `vitest`.17

TypeScript

```ts
// src/features/search/tests/search.store.test.ts

import { describe, it, expect } from "vitest";
import { createRoot, onCleanup } from "solid-js";
import { createSearchStore, Note } from "../search.store";

const mockNotes: Note = },
  { id: "2", title: "Tauri", content: "Build desktop apps", tags: },
  { id: "3", title: "Rust Lang", content: "Performant et sûr", tags: },
];

describe("createSearchStore", () => {
  it("devrait initialiser avec un état par défaut correct", () => {
    createRoot((dispose) => {
      const { state } = createSearchStore();
      expect(state.query).toBe("");
      expect(state.allNotes).toEqual();
      expect(state.isLoading).toBe(false);
      dispose();
    });
  });

  it("devrait filtrer les notes en fonction de la requête", () => {
    createRoot((dispose) => {
      const { actions, filteredNotes } = createSearchStore(mockNotes);

      // Test initial
      expect(filteredNotes()).toHaveLength(3);

      // Action: définir une requête
      actions.setQuery("réactif");

      // Assertion: les notes sont filtrées
      expect(filteredNotes()).toHaveLength(1);
      expect(filteredNotes().title).toBe("SolidJS");

      // Action: vider la requête
      actions.clearSearch();
      expect(filteredNotes()).toHaveLength(3);

      dispose();
    });
  });
```

```
  it("devrait mettre à jour la liste complète des notes", () => {
    createRoot((dispose) => {
      const { state, actions } = createSearchStore();
      expect(state.allNotes).toHaveLength(0);

      actions.loadNotes(mockNotes);

      expect(state.allNotes).toHaveLength(3);
      expect(state.allNotes.title).toBe("Tauri");

      dispose();
    });
  });
});
```

Ce test valide la logique du store en isolation complète, assurant que les actions modifient correctement l'état et que les états dérivés sont calculés comme attendu.17 L'utilisation de

`createRoot` est nécessaire pour fournir un contexte réactif dans lequel les primitives de SolidJS peuvent s'exécuter.

3.1.2. Tester le Composant (`SearchBar.test.tsx`)

Les tests de composants, ou tests d'intégration, valident que l'interface utilisateur interagit correctement avec le store. `@solidjs/testing-library` est l'outil de choix pour cela. Il permet de rendre les composants dans un environnement de test et de simuler les interactions de l'utilisateur.

TypeScript

```
// src/features/search/tests/SearchBar.test.tsx

import { render, screen } from "@solidjs/testing-library";
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";
import { Component, For } from "solid-js";
import { SearchProvider, useSearch } from "../SearchProvider";
import { SearchBar } from "../components/SearchBar";

// Composant de test pour afficher les résultats filtrés
const TestNoteList: Component = () => {
```

```
  const { filteredNotes } = useSearch();
  return (
    <div data-testid="note-list">
      <For each={filteredNotes()}>{(note) => <div>{note.title}</div>}</For>
    </div>
  );
};

const mockNotes = },
  { id: "2", title: "Note sur Tauri", content: "", tags: },
];

describe("SearchBar Integration", () => {
  it("devrait filtrer la liste des notes lorsque l'utilisateur tape dans la
barre de recherche", async () => {
    const user = userEvent.setup();

    // Rendre les composants dans le contexte du Provider
    render(() => (
      <SearchProvider initialNotes={mockNotes}>
        <SearchBar />
        <TestNoteList />
      </SearchProvider>
    ));

    const searchInput = screen.getByLabelText("Barre de recherche");
    const noteList = screen.getByTestId("note-list");

    // État initial
    expect(searchInput).toHaveValue("");
    expect(noteList.children.length).toBe(2);

    // Simuler la saisie de l'utilisateur
    await user.type(searchInput, "Solid");

    // Vérifier que l'état et l'UI ont été mis à jour
    expect(searchInput).toHaveValue("Solid");
    expect(noteList.children.length).toBe(1);
    expect(screen.getByText("Note sur Solid")).toBeInTheDocument();
    expect(screen.queryByText("Note sur Tauri")).not.toBeInTheDocument();
  });
});
```

Ce test valide le flux complet : l'utilisateur tape dans l'input, le composant appelle une action du store, le store met à jour son état, l'état dérivé `filteredNotes` est recalculé, et le composant `TestNoteList` se met à jour pour n'afficher que les résultats pertinents.16

3.1.3. Considérations sur Tauri

Tauri offre ses propres capacités de test, notamment un runtime "mock" pour les tests unitaires qui interagissent avec les API Tauri, et un support pour les tests de bout en bout (E2E) via WebDriver.19 Ces outils sont précieux pour tester l'intégration avec le backend Rust (par exemple, vérifier que l'export vers Obsidian appelle la bonne commande Tauri) ou pour des tests E2E complets de l'application. Cependant, pour la protection des fonctionnalités frontend contre les régressions logiques et d'intégration, les tests unitaires et d'intégration avec

`vitest` et `testing-library` constituent la défense principale et la plus efficace.

3.2. Couche 2 : Sécurité de Type et Documentation Intégrée (Le Contrat Renforcé par le Compilateur et l'Humain)

3.2.1. TypeScript comme Filet de Sécurité

Les interfaces TypeScript définies pour les stores et les `props` des composants ne sont pas de simples aides au développement. Elles constituent un puissant outil automatisé pour prévenir les régressions.12 Si la signature d'une action dans

`search.store.ts` est modifiée (par exemple, en changeant le type d'un argument), le compilateur TypeScript générera une erreur à chaque endroit où cette action est appelée de manière incorrecte. Cela permet de détecter une classe entière de bogues au moment de la compilation, bien avant que le code ne soit exécuté.

3.2.2. JSDoc pour la Clarté

Pour répondre à la demande de "documentation technique intégrée", la pratique la plus efficace est d'adopter une politique stricte d'utilisation des commentaires JSDoc pour chaque fonction, type, et action de store exportés.

TypeScript

```
// src/features/search/search.store.ts (extrait avec JSDoc)

/**
 * Met à jour la requête de recherche dans le store.
 * @param query La nouvelle chaîne de caractères pour la recherche.
 */
setQuery(query: string) {
  setState("query", query);
},
```

Cette approche présente deux avantages majeurs :

1. **Proximité :** La documentation se trouve juste à côté du code qu'elle décrit, ce qui augmente considérablement la probabilité qu'elle soit maintenue à jour lors des refactorings.
2. **Intégration IDE :** Les éditeurs de code modernes comme VS Code utilisent ces commentaires JSDoc pour fournir une autocomplétion et des infobulles enrichies, améliorant l'expérience de développement et facilitant l'onboarding.

3.3. Couche 3 : Gestion de Version Disciplinée (Le Contrat Procédural)

La gestion de version est le filet de sécurité ultime, permettant de comprendre l'historique des changements et de récupérer d'erreurs catastrophiques.

3.3.1. Analyse du Versionnement par Composant

L'idée d'utiliser des `git tags` par composant, bien qu'intéressante, introduit une complexité de gestion significative, surtout pour un développeur seul. Des outils comme les sous-modules Git (`git submodules`) ou les monorepos gérés avec Lerna ou Nx sont conçus pour des équipes gérant des bibliothèques de composants partagés entre plusieurs projets.21 Pour une application unique, même si elle est

modulaire, le coût de maintenance de cette approche dépasse largement ses bénéfices.

3.3.2. Flux de Travail Recommandé

Un flux de travail plus pragmatique et tout aussi puissant est recommandé :

1. **Conventional Commits :** Adopter la spécification Conventional Commits (par exemple, `feat(search): add real-time filtering logic`, `fix(export): handle empty notes correctly`). Cette convention crée un historique de commits lisible à la fois par les humains et les machines.22
2. **Semantic Versioning pour l'Application :** Utiliser les `git tags` (par exemple, `v1.2.0`) pour marquer les versions stables de l'application *dans son ensemble*.

Ce flux de travail offre des outils de lutte contre la régression extrêmement efficaces sans surcharger le développeur :

- **Traçabilité :** Pour comprendre toutes les modifications apportées à la fonctionnalité de recherche, il suffit d'exécuter `git log --grep="\(search\)"`.
- **Récupération :** Si une fonctionnalité entière est accidentellement supprimée ou cassée, la commande `git checkout <commit_hash> -- src/features/feature-name/` permet de restaurer l'intégralité du module de la fonctionnalité à un état antérieur fonctionnel.
- **Diagnostic :** `git bisect` devient un outil chirurgical pour identifier le commit exact qui a introduit une régression, en se basant sur l'historique clair fourni par les commits conventionnels.

Ensemble, ces trois couches — tests automatisés, typage strict et gestion de version disciplinée — forment une stratégie de protection complète. Si un bogue échappe au typage statique, les tests devraient le détecter. Si un bogue complexe passe à travers les tests, l'historique de version clair et la structure modulaire permettent de le localiser et de le corriger rapidement. Cette approche holistique est bien plus résiliente que de se fier à une seule de ces méthodes.

Section 4: Feuille de Route pour la Migration et la Maintenance à Long Terme

Cette dernière section fournit un plan d'action pour migrer de l'architecture monolithique actuelle vers le modèle recommandé, et explique comment cette nouvelle structure répond directement aux préoccupations de maintenance à long terme.

4.1. Chemin de Migration Incrémentiel depuis `App.tsx`

La refonte d'un monolithe est une opération risquée si elle est entreprise en une seule fois. Une approche incrémentale, fonctionnalité par fonctionnalité, est plus sûre et permet de valider le nouveau modèle progressivement.

- **Étape 1 : Identifier la Plus Petite Fonctionnalité.** Commencer par la fonctionnalité la moins complexe et la plus isolée, par exemple, le "Modal d'export". Cela permet de se familiariser avec le processus de refactoring et de valider le modèle "Feature-Store" avec un risque minimal.
- **Étape 2 : Créer le Module de Fonctionnalité.** Mettre en place la structure de répertoires décrite dans la Section 2 pour la fonctionnalité choisie (par exemple, `src/features/export/`).
- **Étape 3 : Extraire et Isoler le Composant.** Déplacer soigneusement le code JSX du modal d'export depuis `App.tsx` vers son nouveau fichier de composant (par exemple, `src/features/export/components/ExportModal.tsx`).
- **Étape 4 : Créer le Store.** Identifier tout l'état (`createSignal`) et toute la logique (gestionnaires d'événements, appels à l'API Tauri) liés à l'export dans `App.tsx`. Extraire cette logique et la placer dans un nouveau store (`src/features/export/export.store.ts`) en utilisant `createStore`.
- **Étape 5 : Fournir et Consommer le Contexte.** Créer le `ExportProvider.tsx`. Dans `App.tsx`, remplacer l'ancien composant de modal par le nouveau, enveloppé dans son `ExportProvider`. Refactoriser le composant `ExportModal.tsx` pour qu'il utilise `useContext` (`useExport()`) afin d'accéder à son état et à ses actions.
- **Étape 6 : Tester.** Écrire des tests unitaires pour le nouveau `export.store.ts` et des tests d'intégration pour `ExportModal.tsx`. Ces tests doivent valider que la fonctionnalité n'a subi aucune régression pendant le refactoring. C'est une étape de validation critique.
- **Étape 7 : Nettoyer `App.tsx`.** Une fois que la fonctionnalité est entièrement migrée et que les tests passent, supprimer en toute

confiance l'ancien code (état, logique, JSX) de `App.tsx`. Le monolithe a commencé à rétrécir.
- **Étape 8 : Répéter.** Appliquer ce même processus pour les autres fonctionnalités, une par une (Recherche, Catégories, WikiLinks), en s'attaquant progressivement à des fonctionnalités plus complexes. Chaque itération rend l'application plus modulaire et plus robuste.

4.2. Répondre aux Questions Fondamentales de Maintenance

Cette nouvelle architecture apporte des réponses directes et claires aux questions de maintenance posées initialement.

4.2.1. Sur la Récupération de Composants Supprimés Accidentellement

Avec l'ancienne architecture monolithique, la suppression accidentelle d'une fonctionnalité impliquait de démêler des lignes de code interconnectées dans un fichier massif. Avec le modèle "Feature-Store", chaque fonctionnalité est un module autonome et co-localisé dans le système de fichiers. Grâce à l'historique Git discipliné, la récupération devient une opération simple et ciblée. La commande `git checkout <hash_du_dernier_commit_fonctionnel> -- src/features/ma-fonctionnalite/` restaure l'intégralité du répertoire de la fonctionnalité — sa logique, son interface utilisateur et ses tests — sans affecter le reste de l'application. L'historique clair fourni par les Commits Conventionnels facilite grandement l'identification du bon commit à restaurer.

4.2.2. Sur l'Évolution Indépendante des Fonctionnalités

La principale faiblesse d'un monolithe est le fort couplage : une modification dans une partie du code peut avoir des effets de bord imprévus ailleurs. Le modèle "Feature-Store" résout ce problème en créant des frontières claires entre les fonctionnalités. Le store et le contexte agissent comme une interface publique bien définie pour chaque module. Tant que cette interface (les types d'état et les signatures d'actions) est respectée, l'implémentation interne d'une fonctionnalité peut être modifiée, améliorée ou même complètement

réécrite avec un risque minimal d'impacter les autres parties de l'application. Une fonctionnalité peut évoluer à son propre rythme, ce qui est essentiel pour une maintenance agile et à long terme.

4.2.3. Sur l'Onboarding de Nouveaux Développeurs

L'un des plus grands défis d'un monolithe est la charge cognitive qu'il impose. Un nouveau développeur doit comprendre l'ensemble du système avant de pouvoir apporter la moindre modification en toute sécurité. Le modèle "Feature-Store" offre une solution élégante à ce problème. Pour confier à un nouveau développeur la tâche de corriger un bogue ou d'ajouter une amélioration à la fonctionnalité de recherche, il suffit de le pointer vers le répertoire `src/features/search/`. Ce répertoire contient une "tranche verticale" complète et autonome de l'application :

- `search.store.ts` : Comment la fonctionnalité fonctionne (la logique).
- `components/` : À quoi elle ressemble (l'interface).
- `tests/` : Comment elle est censée se comporter (les spécifications).

Le développeur peut se concentrer sur ce périmètre limité, comprendre rapidement le contexte et devenir productif en un temps record, sans avoir besoin de déchiffrer les complexités d'un `App.tsx` de plusieurs milliers de lignes. Une bonne architecture n'est pas seulement techniquement performante ; elle est aussi un outil qui facilite la cognition humaine. En décomposant un problème vaste et complexe (l'application) en une série de problèmes plus petits, indépendants et compréhensibles (les fonctionnalités), on améliore directement tous les aspects de la maintenance : le débogage, l'évolution et la collaboration.

Section 5: Synthèse Stratégique et Recommandations

Ce rapport a analysé les défis posés par une architecture frontend monolithique dans le contexte d'une application de bureau performante développée avec Tauri et SolidJS. Face aux risques de régression fonctionnelle et aux difficultés de maintenance, une transition vers une architecture modulaire est impérative.

L'analyse des différentes options de gestion d'état dans SolidJS a révélé que ni les composants autonomes gérés par `props`, ni les hooks

personnalisés ne sont adaptés à la gestion d'état partagé complexe. La solution réside dans l'utilisation des **stores** de SolidJS.

La recommandation principale est l'adoption d'un **modèle architectural hybride "Feature-Store"**. Ce modèle structure l'application par fonctionnalités métier plutôt que par couches techniques. Chaque fonctionnalité est encapsulée dans son propre module, contenant :

1. Un **store dédié** (`createStore`) qui gère l'état, les états dérivés et la logique métier de la fonctionnalité.
2. Des **composants légers** qui consomment le store via un **contexte** (`createContext` / `useContext`).
3. Des **tests co-localisés** qui valident la logique et l'intégration de la fonctionnalité.

Pour protéger l'application contre les régressions, une **stratégie de défense multi-couches** est préconisée :

1. **Tests Automatisés :** Des tests unitaires pour les stores (`vitest`) et des tests d'intégration pour les composants (`@solidjs/testing-library`) afin de valider le comportement à chaque modification.
2. **Typage Strict et Documentation :** L'utilisation de TypeScript et de JSDoc pour créer des contrats clairs, renforcés par le compilateur, qui préviennent les erreurs d'intégration.
3. **Gestion de Version Disciplinée :** L'adoption des Commits Conventionnels et du versionnement sémantique pour l'application, offrant une traçabilité complète et des mécanismes de récupération robustes via Git.

Enfin, une **feuille de route de migration incrémentale** a été proposée pour passer en toute sécurité du monolithe `App.tsx` à cette nouvelle architecture, en commençant par la fonctionnalité la moins complexe pour minimiser les risques.

En conclusion, le modèle "Feature-Store", combiné à une stratégie de protection multi-couches, représente l'approche la plus efficace pour construire une application Tauri/SolidJS robuste, performante et maintenable. Cette architecture est non seulement capable de s'adapter aux exigences futures, mais elle est également parfaitement gérable et structurante pour un développeur principal, en réduisant la charge cognitive et en favorisant des pratiques de développement saines et durables.