Rapport d'analyse de l'application RPG

Phase 3: Examen de la logique métier (IA Narrative et Génération d'images)

1. `llm-manager.ts` : - **Incohérence:** La méthode `initDefaultSettings` initialise les clés API à partir des variables d'environnement (`import.meta.env`), mais les commentaires indiquent que `saveSettings` (qui persisterait les paramètres dans `localStorage`) n'est plus nécessaire car les fichiers `.env` sont la "source de vérité". Cependant, `illustration-generator.ts` tente de lire les clés API depuis `localStorage` pour OpenRouter, DeepSeek et Grok. Cela crée une incohérence dans la gestion des clés API. Les clés API devraient être gérées de manière cohérente, soit uniquement via les variables d'environnement, soit via `localStorage` (avec une initialisation par défaut depuis les variables d'environnement si `localStorage` est vide). - **Amélioration:** La méthode `getNarrativeAI()` appelle `this.applySettings()` à chaque fois. Bien que cela garantisse que les paramètres les plus récents sont appliqués, cela pourrait être optimisé si les paramètres ne changent pas fréquemment. Une vérification pour voir si les paramètres ont changé pourrait être ajoutée pour éviter des appels inutiles à `setApiProvider` .

2. `advanced-narrative-ai.ts` : - **Logique de Fallback:** La logique de fallback dans `generateResponse` semble correcte. Si `apiProvider` ou `apiKey` ne sont pas définis, elle utilise `generateFallbackResponse()` . Cependant, la gestion des erreurs (`catch (apiError)`) ne tente pas d'autres fournisseurs et utilise directement le fallback. Cela est intentionnel (`Ne pas essayer d'autres fournisseurs, utiliser directement le fallback pour éviter la confusion dans les logs`), mais pourrait être une limitation si l'on souhaite une résilience accrue en cas de défaillance d'un fournisseur. - **Modèles par défaut:** Les modèles par défaut définis dans `setApiProvider` sont corrects et correspondent aux fournisseurs.

3. `CharacterCreation.tsx` : - **Gestion des clés API:** Ce composant utilise `llmManager.getNarrativeAI()` pour obtenir l'instance de l'IA narrative, ce qui est correct. Il ne gère pas directement les clés API, s'appuyant sur `llmManager` . - **Génération de personnage:** La logique de construction du prompt pour la génération de personnage est robuste, incluant les informations déjà saisies et un `randomSeed` pour la variété. L'extraction du JSON de la réponse de l'IA est également bien gérée avec un fallback si le bloc `json` n'est pas trouvé. - **Génération d'avatar:** La fonction `generateAvatar()` est appelée après la génération du personnage, ce qui est une

bonne pratique. Le prompt pour l'avatar est construit à partir des données du personnage, ce qui assure la cohérence.

4. `illustration-generator.ts` : - **Incohérence de gestion des clés API:** Comme mentionné précédemment, cette classe tente de récupérer les clés API pour OpenRouter, DeepSeek et Grok depuis `localStorage` (`llmSettingsJson = localStorage.getItem('llm-settings')`). Cela contredit la logique de `llm-manager.ts` qui indique que les clés API sont gérées via les variables d'environnement et non persistées dans `localStorage`. Cela pourrait entraîner des problèmes si les clés API ne sont pas définies dans `localStorage` mais sont présentes dans les variables d'environnement. - **Priorité du fournisseur:** Le fournisseur par défaut est Hugging Face, ce qui est cohérent avec les exigences précédentes. Cependant, la logique de récupération des clés API pour les autres fournisseurs depuis `localStorage` est une source potentielle d'erreur. - **Fallback d'image:** La fonction `getFallbackImage` est une bonne implémentation pour gérer les échecs de génération d'images.

Phase 4: Vérification de l'interface utilisateur et de la navigation

1. `HomePage.tsx` : - **Navigation:** La navigation vers les différentes sections (Nouvelle Aventure, Charger une Aventure, Créer un Personnage) est gérée via des composants `<Link>` de `react-router-dom`, ce qui est une approche standard et fonctionnelle. - **Modals:** Les modales pour les paramètres LLM et les paramètres de jeu sont correctement implémentées et gérées par des états locaux (`showLLMSettings`, `showGameSettings`). - **Affichage de la configuration LLM:** Le composant affiche correctement le statut de la configuration LLM (fournisseur actif, modèle, et si une clé API est configurée), ce qui est utile pour l'utilisateur.

2. `GameInterface.tsx` : - **Bouton Retour:** Un bouton "Retour" a été ajouté dans l'en-tête, utilisant `navigate(-1)` pour revenir à la page précédente. C'est une implémentation simple et efficace pour la navigation arrière. - **Navigation contextuelle:** La navigation vers le combat (`/combat`) est gérée via `navigate` avec un `state` pour passer les données du joueur, l'univers et le style narratif, ce qui est approprié pour maintenir le contexte du jeu. - **Défilement des messages:** L'utilisation de `messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });` assure que les nouveaux messages sont visibles, améliorant l'expérience utilisateur.

3. `CharacterCreation.tsx` : - **Navigation après création:** Après la création d'un personnage, l'application navigue vers `/game`, ce qui est logique. Le `navigate` hook est utilisé correctement. - **Formulaire:** Le formulaire de création de personnage est bien structuré avec des champs de saisie, des sélecteurs et des cases à cocher pour les traits

et compétences. La gestion des changements (`handleChange` , `handleTraitChange` , `handleSkillChange`) est standard pour React.

Conclusion de la Phase 4: L'interface utilisateur et la navigation semblent bien conçues et fonctionnelles. Les éléments de navigation sont cohérents et les interactions utilisateur sont gérées de manière appropriée. Le bouton retour a été ajouté et fonctionne comme prévu.

Phase 5: Analyse de la sécurité et des bonnes pratiques

1. Gestion des clés API (`.env.local` et code): - **Exposition des clés API:** Les clés API sont stockées dans `.env.local` et chargées via `import.meta.env.VITE_...` . Bien que cela soit une pratique courante pour les applications React/Vite, il est crucial de comprendre que ces variables d'environnement préfixées par `VITE_` sont **exposées côté client** après la compilation. Cela signifie que n'importe qui peut inspecter le code source de l'application déployée et récupérer ces clés API. C'est une **vulnérabilité majeure**. - **Recommandation:** Pour les clés API sensibles (comme celles des LLM ou des générateurs d'images), elles ne devraient **jamais** être exposées côté client. Elles devraient être gérées par un backend sécurisé (par exemple, une fonction serverless, un proxy API) qui effectue les appels aux services externes. L'application frontend appellerait alors ce backend, qui lui-même utiliserait les clés API de manière sécurisée côté serveur. - **Incohérence persistante:** L'incohérence notée dans la phase 3 concernant la lecture des clés API depuis `localStorage` dans `illustration-generator.ts` alors que `llm-manager.ts` les charge depuis les variables d'environnement est un problème de cohérence et de sécurité. Si `localStorage` est utilisé, les clés y seraient également exposées. La solution recommandée est de centraliser la gestion des clés côté serveur.

2. Content Security Policy (CSP) (`public/_headers`): - **Amélioration:** La CSP a été mise à jour pour inclure `https://plausible.io` et `https://*.amplitude.com` pour les scripts et les connexions. C'est une bonne pratique pour autoriser uniquement les sources de confiance. - **`unsafe-inline` et `unsafe-eval` :** L'utilisation de `'unsafe-inline'` pour `script-src` et `style-src` , et `'unsafe-eval'` pour `script-src` est une **vulnérabilité potentielle**. Ces directives permettent l'exécution de code JavaScript et de styles CSS inline, ce qui peut ouvrir la porte aux attaques de Cross-Site Scripting (XSS) si une injection de contenu est possible. Idéalement, ces directives devraient être évitées en favorisant des hachages (hashes) ou des nonces (nonces) pour les scripts et styles inline, ou en déplaçant tout le code dans des fichiers externes. - **`img-src` :** L'inclusion de `data:` et `https:` est généralement acceptable pour les images, permettant les images encodées en base64 et les images externes. - **`connect-src` :** La liste des domaines autorisés pour les connexions (`connect-src`)

est exhaustive et semble couvrir tous les services externes utilisés (Manuss, OpenRouter, DeepSeek, Grok, Supabase, Hugging Face, Plausible, Amplitude). C'est une bonne pratique pour limiter les connexions non autorisées.

3. Gestion des erreurs et logs: - L'application utilise `console.error` et `console.warn` pour les erreurs et avertissements, ce qui est utile pour le débogage. Cependant, dans un environnement de production, il serait préférable d'utiliser un système de logging plus robuste qui envoie les erreurs à un service de surveillance (ex: Sentry, LogRocket) plutôt que de simplement les afficher dans la console du navigateur.

4. Authentification et Autorisation: - Le code mentionne `user_id: 'local-user'` qui serait remplacé par un vrai ID si en mode Supabase. Il est crucial de s'assurer que l'authentification et l'autorisation sont correctement implémentées côté serveur pour protéger les données des utilisateurs et prévenir les accès non autorisés, surtout si des données sensibles sont stockées dans Supabase.

Conclusion de la Phase 5: L'application a fait des efforts pour implémenter une CSP et gérer les clés API via des variables d'environnement. Cependant, l'exposition des clés API côté client et l'utilisation de `unsafe-inline` / `unsafe-eval` dans la CSP sont des points de sécurité critiques qui nécessitent une attention immédiate. Une refonte de l'architecture pour gérer les clés API côté serveur est fortement recommandée.

Conclusion Générale

L'application RPG "Chroniques d'Étheria" présente une architecture bien structurée et une logique métier globalement cohérente pour ses fonctionnalités principales (IA narrative, génération d'images, création de personnage). Les efforts pour améliorer la navigation et la gestion des paramètres sont notables.

Cependant, des points critiques ont été identifiés, principalement liés à la **sécurité et à la gestion des clés API**. L'exposition des clés API côté client est une vulnérabilité majeure qui doit être adressée en priorité par une refonte de l'architecture pour gérer ces clés côté serveur. De même, l'utilisation de directives `unsafe-inline` et `unsafe-eval` dans la Content Security Policy devrait être réévaluée et renforcée pour minimiser les risques de XSS.

Des incohérences dans la gestion des clés API entre `llm-manager.ts` et `illustration-generator.ts` ont également été relevées, soulignant la nécessité d'une approche unifiée et sécurisée pour la configuration des services externes.

En corrigeant ces vulnérabilités et en optimisant la gestion des dépendances, l'application pourra offrir une expérience plus robuste, sécurisée et maintenable.

Comparaison de l'architecture actuelle avec l'architecture des 4 moteurs

L'architecture actuelle de l'application RPG est plus monolithique, avec une logique principalement centrée autour de `AdvancedNarrativeAI` pour la gestion des interactions et de la narration, et `IllustrationGenerator` pour la génération d'images. Elle ne sépare pas explicitement les rôles en quatre moteurs distincts comme proposé dans l'architecture cible.

Voici une comparaison détaillée :

Moteur 1 : Interaction & Parsing (Le Contrôleur)

- **Architecture Actuelle :** La logique de parsing des actions du joueur est implicitement gérée au sein de `GameInterface.tsx` et `AdvancedNarrativeAI`. `GameInterface.tsx` prend l'entrée utilisateur et la passe à `AdvancedNarrativeAI` qui, à son tour, génère une réponse narrative. Il n'y a pas de moteur de parsing dédié qui catégorise et formate les actions du joueur en JSON structuré avant de les passer à d'autres systèmes.
- **Architecture Cible :** Un moteur dédié qui interprète l'action du joueur, détermine les systèmes impactés et formate la réponse en JSON structuré. Cela permet une standardisation des entrées pour les autres moteurs et une meilleure orchestration.
- **Écart :** Important. L'application actuelle manque d'une couche de parsing et d'orchestration explicite.

Moteur 2 : État Global (La Base de Données Vivante)

- **Architecture Actuelle :** L'état du jeu (messages, personnage, univers, style narratif) est géré via des états React (`useState`) dans `GameInterface.tsx` et via `localStorage` pour la persistance (`saveContextToLocalStorage` , `loadContextFromLocalStorage` dans `AdvancedNarrativeAI`). `AdvancedNarrativeAI` maintient un historique des messages pour son contexte. Il n'y a pas de source de vérité unique et centralisée sous forme de base de données vivante qui gère la cohérence, la compression contextuelle et l'extraction ciblée pour d'autres moteurs.
- **Architecture Cible :** Un moteur dédié qui agit comme une source de vérité unique pour l'état du monde et du joueur, gérant la cohérence, la compression contextuelle et fournissant des extraits pertinents aux autres moteurs. L'utilisation de bases de données (SQLite/PostgreSQL) et de cache (Redis) est recommandée.

- **Écart :** Important. La gestion de l'état est distribuée et moins structurée que l'approche proposée. La compression contextuelle et le rollback ne sont pas implémentés.

Moteur 3 : Narration & Dialogue (Le Conteur)

- **Architecture Actuelle :** `AdvancedNarrativeAI` remplit ce rôle. Il génère la narration et les dialogues en se basant sur l'historique des messages et les paramètres narratifs (univers, style). Il est capable de produire des descriptions immersives. Cependant, il n'y a pas de spécialisation explicite pour le dialogue avancé avec mémoire ou la gestion émotionnelle au-delà de ce que le LLM peut inférer du prompt.
- **Architecture Cible :** Un moteur spécialisé dans la génération narrative, les dialogues PNJ avec mémoire et personnalité, et les descriptions environnementales. Il reçoit des actions parsées et un état pertinent du monde.
- **Écart :** Modéré. `AdvancedNarrativeAI` est le composant le plus proche de ce moteur, mais il pourrait bénéficier d'une meilleure intégration avec un moteur d'état global et de fonctionnalités plus avancées pour la gestion des PNJ et de l'ambiance.

Moteur 4 : Simulation Mondiale (L'Univers Vivant)

- **Architecture Actuelle :** Il n'y a pas de moteur de simulation mondiale explicite. Les événements sont principalement déclenchés par les actions du joueur ou des transitions d'univers manuelles. Il n'y a pas de logique pour générer des événements

autonomes ou des conséquences à long terme en dehors de l'interaction directe du joueur. - **Architecture Cible :** Un moteur qui génère des événements autonomes, fait évoluer le monde et gère les conséquences à long terme, déclenché périodiquement ou par des événements spécifiques. - **Écart :** Important. Cette fonctionnalité est largement absente de l'application actuelle.

Résumé des Écarts

L'application actuelle est une application web frontend qui utilise des LLM pour la narration et la génération d'images. Elle est principalement axée sur l'interaction directe du joueur avec l'IA narrative. L'architecture des 4 moteurs propose une approche beaucoup plus modulaire et distribuée, avec des responsabilités clairement séparées pour le parsing des actions, la gestion de l'état global, la narration et la simulation mondiale. Les principaux écarts résident dans l'absence de moteurs dédiés pour

l'interaction/parsing et la simulation mondiale, ainsi qu'une gestion de l'état global moins structurée et centralisée.

La transformation vers l'architecture des 4 moteurs nécessiterait une refonte significative, notamment l'introduction d'un backend pour gérer les moteurs d'état et de simulation, et pour centraliser la gestion des clés API de manière sécurisée.

Plan de Transformation vers l'Architecture des 4 Moteurs

La transformation de l'application actuelle vers l'architecture des 4 moteurs est un projet significatif qui nécessitera une refonte architecturale majeure, notamment l'introduction d'un backend. Voici un plan en plusieurs phases :

Phase 1 : Introduction d'un Backend Sécurisé et Moteur d'Interaction

- **Objectif :** Mettre en place un backend pour gérer les appels API sensibles et implémenter le Moteur d'Interaction.
- **Tâches :**
  - **1.1. Création du Backend :** Mettre en place un framework backend (ex: Flask, FastAPI) pour servir de passerelle sécurisée pour les appels LLM et de génération d'images. Cela résoudra le problème de l'exposition des clés API côté client.
  - **1.2. Migration des Appels API :** Déplacer les appels aux API LLM (DeepSeek, OpenRouter, Grok, Gemini) et à l'API Hugging Face du frontend vers le nouveau backend. Le frontend communiquera avec le backend via des endpoints sécurisés.
  - **1.3. Implémentation du Moteur d'Interaction :** Développer le Moteur d'Interaction côté backend. Ce moteur recevra les actions brutes du joueur du frontend, les interprétera, déterminera les systèmes impactés et formatera une réponse JSON structurée.
  - **1.4. Mise à jour du Frontend :** Adapter le frontend pour qu'il envoie les actions du joueur au nouveau backend et reçoive les réponses structurées du Moteur d'Interaction.

Phase 2 : Développement du Moteur d'État Global

- **Objectif :** Centraliser la gestion de l'état du jeu et du monde.

- **Tâches :**
  - **2.1. Conception de la Base de Données d'État :** Définir le schéma de données pour stocker l'état global du monde, du joueur, des quêtes, des PNJ, etc. (ex: PostgreSQL).
  - **2.2. Implémentation du Moteur d'État :** Développer le Moteur d'État côté backend. Ce moteur sera la source de vérité unique, gérant les mises à jour, la cohérence, la compression contextuelle et l'extraction ciblée des données.
  - **2.3. Intégration avec le Moteur d'Interaction :** Le Moteur d'Interaction communiquera avec le Moteur d'État pour obtenir et mettre à jour les informations pertinentes.
  - **2.4. Migration de la Persistance :** Migrer la logique de sauvegarde/chargement du contexte et des messages de `localStorage` vers le Moteur d'État et sa base de données.

Phase 3 : Affinement du Moteur de Narration et Introduction du Moteur de Simulation

- **Objectif :** Améliorer la narration et introduire la dynamique du monde autonome.
- **Tâches :**
  - **3.1. Refonte du Moteur de Narration :** Adapter `AdvancedNarrativeAI` (ou son équivalent backend) pour qu'il s'intègre pleinement avec le Moteur d'État. Il recevra des données structurées du Moteur d'État et produira une narration plus riche et cohérente.
  - **3.2. Implémentation du Moteur de Simulation :** Développer le Moteur de Simulation côté backend. Ce moteur générera des événements mondiaux autonomes, fera évoluer les factions, les PNJ et les quêtes en arrière-plan, et mettra à jour le Moteur d'État en conséquence.
  - **3.3. Déclencheurs de Simulation :** Mettre en place les déclencheurs pour le Moteur de Simulation (ex: toutes les X actions du joueur, après un repos long, changement de zone).

Phase 4 : Optimisations et Extensions

- **Objectif :** Améliorer les performances, la résilience et ajouter des fonctionnalités avancées.
- **Tâches :**
  - **4.1. Cache Intelligent :** Implémenter des mécanismes de cache (ex: Redis) pour les réponses narratives récurrentes, les deltas d'état, et la persistance des PNJ.
  - **4.2. Modèles LLM Hybrides :** Mettre en place une logique pour utiliser différents modèles LLM (moins chers pour le parsing simple, plus performants

pour la narration complexe) en fonction des besoins spécifiques de chaque moteur.
    - **4.3. Métriques et Monitoring :** Intégrer des outils de monitoring pour suivre l'utilisation des tokens, les temps de réponse, l'engagement des joueurs et l'efficacité de la compression contextuelle.
    - **4.4. Extensions Futures :** Préparer l'architecture pour des extensions comme le mode multi-joueur, la génération procédurale ou l'analyse comportementale.

Ce plan de transformation permettra à l'application de bénéficier d'une architecture plus robuste, sécurisée, scalable et modulaire, offrant une expérience de jeu plus riche et dynamique.