Web Scraper & Générateur IA

Ce projet est une application web développée avec Vue.js permettant d'extraire des données de Wikipedia et de générer du nouveau contenu à partir de ces informations via un serveur Ollama local. Pour dire vrai, Ce projet a été essentiellement développé avec l'aide de Claude Sonnet. Toutefois, l'objectif est de reproduire progressivement toutes les fonctionnalités essentielles à partir de cette base technique, que j'ai déjà explorée pendant de nombreuses heures, tout en ajoutant de nouvelles fonctionnalités et améliorations à l'avenir.

Objectif du projet

L'objectif principal de ce projet est de créer une application web qui:

1. Extrait (scrape) des données depuis des pages Wikipédia
2. Utiliser ces données comme base pour générer de nouveaux contenus via un modèle d'IA local
3. Offre une expérience utilisateur fluide et intuitive
4. Permet de conserver un historique des extractions et générations

Fonctionnalités principales

Extraction de données (Scraping)

- Extraction ciblée des titres et paragraphes de pages Wikipedia
- Affichage visuel des données extraites
- Historique des extractions précédentes
- Interface de détail pour visualiser la structuration des données extraites

Génération de contenu avec IA

- Utilisation du modèle Phi (ou autres modèles) via Ollama pour générer du contenu
- Prompts personnalisables avec suggestions
- Sauvegarde automatique des générations
- Affichage formaté des résultats générés

Interface utilisateur

- Design responsive
- Navigation intuitive entre les différentes fonctionnalités
- Aperçus visuels des données extraites et générées
- Gestion complète des états de chargement et des erreurs

Stack technique

- **Frontend**: Vue.js 3 avec Composition API
- **État**: Pinia pour la gestion d'état
- **Routing**: Vue Router
- **HTTP**: Axios pour les requêtes API
- **IA**: Intégration avec Ollama (modèle Phi)
- **Persistence**: LocalStorage pour sauvegarder les données localement

Installation et démarrage

Prérequis

- Node.js (v14+)
- npm ou yarn
- Ollama installé localement (pour la génération IA)

Installation d'Ollama

Windows

1. Téléchargez et installez Ollama depuis https://ollama.com/download/windows
2. Ouvrez une invite de commande et exécutez: `ollama run phi` ou modèle de votre choix
3. Vérifiez que Ollama est en cours d'exécution (icône dans la zone de notification)

macOS

1. Téléchargez et installez Ollama depuis https://ollama.com/download/mac
2. Ouvrez un Terminal et exécutez: `ollama run phi` ou modèle de votre choix
3. Vérifiez que Ollama est en cours d'exécution (icône dans la barre de menu)

Configuration du modèle dans le projet

Dans le fichier `frontend/src/stores/llm.js`, assurez-vous que le nom du modèle correspond exactement à celui que vous avez installé avec Ollama:

```
ollamaConfig: {
  baseUrl: "http://localhost:11434",
  model: "deepseek-r1", // Remplacez par le modèle que vous utilisez :latest peut être nécessaire
},
```

Le nom du modèle doit correspondre exactement à celui affiché dans la commande `ollama list`. Voici quelques exemples courants:

- `phi:latest` - Microsoft Phi
- `deepseek-r1:latest` - DeepSeek Coder R1
- `llama3:latest` - Llama 3
- `mistral:latest` - Mistral
- `gemma:latest` - Gemma

Dépannage Ollama

Si vous rencontrez des problèmes avec Ollama, voici quelques commandes utiles:

Vérifier les modèles installés

```
ollama list
```

Vérifier l'état Ollama

```
curl http://localhost:11434/api/tags
```

Problèmes de connexion refusée

Si vous obtenez une erreur de connexion refusée, vérifiez que:

1. Ollama est bien en cours d'exécution
2. Le port 11434 n'est pas bloqué par un pare-feu
3. Vous utilisez bien `http://localhost:11434` comme URL de base

Téléchargement d'un nouveau modèle

Si vous souhaitez essayer un modèle différent:

```
ollama pull nom_du_modele:latest

ollama run nom_du_modele fonctionne aussi
```

N'oubliez pas de mettre à jour le nom du modèle dans votre code après l'installation d'un nouveau modèle.

Installation du serveur backend pour le scraping

1. Naviguez vers le dossier `backend` du projet

```
cd backend
```

2. Installez les dépendances

```
npm install express puppeteer cors
```

3. Naviguez vers le dossier `backend\src` du projet

```
cd .\src\
```

4. Démarrez le serveur

```
node .\index.js
```

Le serveur sera disponible sur `http://localhost:3000`

Installation du frontend

1. Dans un autre terminal, naviguez vers la racine du projet

```
cd .\frontend\
```

2. Installez les dépendances

```
npm install
```

3. Démarrez l'application

```
npm run dev
```

4. Ouvrez l'application dans votre navigateur à l'adresse indiquée (généralement `http://localhost:5173`)

Difficultés rencontrées et solutions

1. Intégration avec Ollama

Problème: L'intégration d'un LLM a été un véritable défi. À l'origine, je comptais simplement utiliser OpenAI ou un autre LLM pour générer une clé API et effectuer mes appels. Cependant, le coût de ces services a compliqué les choses !

Solution: J'ai donc dû chercher une solution à contrecœur et installer un LLM en local pour effectuer mes appels. Je me suis finalement tourné vers Ollama, sans doute l'un des meilleurs dans ce domaine. Cependant, une fois le texte extrait, la fiabilité de la réponse dépend fortement du modèle utilisé.

2. Structuration des données scrapées

Problème: La mise en relation des titres et paragraphes extraits de Wikipedia pas si simple que ça au final.

Solution: Pour cette partie, j'ai visionné plusieurs vidéos afin de mieux comprendre le fonctionnement général du scraping et mis en place des solutions simples. Cependant, lorsque j'ai voulu contourner certaines classes et créer un parsing efficace, j'ai dû faire appel à Claude Sonnet pour m'aider. Ces deux vidéos ont été très utiles : https://www.youtube.com/watch?v=CosPILI0qrw & https://www.youtube.com/watch?v=ssRo5nVOvrQ

3. Persistence des données

Problème: Garder l'historique des données scrapées et des générations entre les sessions.

Solution: J'ai structuré l'utilisation de localStorage de manière claire pour sauvegarder les états Pinia, avec une logique de chargement au démarrage de l'application. Pour ce projet, j'ai estimé que l'implémentation de Firebase serait trop ambitieuse, alors j'ai préféré opter pour localStorage.

4. Gestion des requêtes asynchrones

Problème : Gestion du chargement et des erreurs lors des opérations asynchrones.

Solution : Ajout des états isLoading et error dans les stores, avec des timeouts pour éviter les requêtes trop longues.

Améliorations futures

- Possibilité d'avoir plusieurs pages wikipedia avant de tout scraper
- Ajouter la possibilité de scraper d'autres sites que Wikipedia
- Implémenter un système de tags pour organiser les générations
- Ajouter des options d'export (PDF, Markdown, etc.)
- Améliorer les suggestions de prompts avec un système d'apprentissage basé sur les prompts utilisés
- Ajouter un mode hors-ligne avec synchronisation ultérieure
- etc ...

Ce projet a été développé par [OUI] pour le cours Framework Web 2.