Documentation technique

Prérequis et configuration

Prérequis

Pour faire fonctionner le projet, assurez-vous d'avoir installé :
- Node.js (20.9.0)
- Docker (25.0.3)
- Angular CLI (17.1)
- NestJS CLI (10.1.18)

Configuration de la base de données

Créez la base de données en exécutant les commandes suivantes :

```
bash
cd backend/scripts
docker-compose up -d # pour lancer la base de données
docker-compose down -v # pour supprimer la base de données
```

Le port par défaut est le 27017.

Lancement du backend

Pour lancer le backend :
1. Allez dans le dossier backend.
2. Exécutez la commande `npm install`.
3. Exécutez la commande `npm start`.
4. Le backend est lancé sur le port 3000.

Lancement du frontend

Pour lancer le frontend :
1. Allez dans le dossier frontend.
2. Exécutez la commande `npm install`.
3. Exécutez la commande `npm start`.
4. Le frontend est lancé sur le port 4200.

Utilisation de conteneurs Docker

Des conteneurs Docker sont disponibles pour le backend, le frontend et la base de données. Pour les lancer, exécutez les commandes suivantes :

```
cd server && docker-compose -f docker-compose.yml up
```

Cela construira trois images Docker et lancera les conteneurs correspondants. Ces conteneurs sont utilisés pour le déploiement de l'application.

Exécution des tests

Pour exécuter les tests, exécutez la commande `npm run test` dans le dossier backend. Pour exécuter les tests en mode watch, exécutez la commande `npm run test:watch`.

Organisation des fichiers

Backend

Voici l'arborescence des fichiers :

```
C:.
│   .dockerignore
│   .env
│   .eslintrc.js
│   .gitignore
│   .prettierrc
│   .puppeteerrc.cjs
│   Dockerfile
│   nest-cli.json
│   package-lock.json
│   package.json
│   tsconfig.build.json
│   tsconfig.json
├───ressources
│   └───sirene
│           NafCodes.json
│           StockEtab.csv
│           StockUL.csv
├───scripts
│       docker-compose.yml
└───src
    │   app.controller.ts
    │   app.module.ts
    │   main.ts
    ├───api
    │   ├───ban
    │   │       ban.service.spec.ts
    │   │       ban.service.ts
    │   └───sirene
    │           scrap-sirene.dto.ts
    │           sirene.service.spec.ts
    │           sirene.service.ts
    ├───entreprise
    │   │   entreprise.module.ts
    │   ├───controllers
    │   │       entreprise.controller.ts
    │   │       fichier.controller.ts
    │   ├───dao
    │   │       entreprise-dao.ts
    │   │       fichier-dao.ts
    │   ├───dto
    │   │       entreprisesids.dto.ts
    │   ├───entities
```

```
│   │       entreprise.entity.ts
│   │       entreprise.Finance.entity.ts
│   │       entreprise.location.entity.ts
│   │       entreprise.representative.entity.ts
│   │       entreprise.review.entity.ts
│   │       fichier.entity.ts
│   ├───schema
│   │       entreprise.schema.ts
│   │       finance.schema.ts
│   │       location.schema.ts
│   │       representative.schema.ts
│   │       review.schema.ts
│   └───service
│           entreprise.service.ts
│           fichier.service.ts
├───parameter
│   │   parameter.module.ts
│   ├───controller
│   │       parameter.controller.ts
│   ├───dao
│   │       parameter.dao.ts
│   ├───dto
│   │       parameter.time.update.ts
│   ├───entity
│   │       parameter.entity.ts
│   ├───schema
│   │       parameter.schema.ts
│   └───service
│           parameter.service.ts
├───scraping
│   ├───entreprise
│   │       pappers.service.ts
│   │       societe.service.ts
│   └───review
│           avis-verifies.service.ts
│           gowork.service.ts
└───sirene-entreprise
    │   naf.module.ts
    │   sirene-entreprise.controller.ts
    │   sirene-entreprise.module.ts
    ├───dao
    │       naf.dao.ts
    │       sirene-entreprise.dao.ts
    ├───entities
    │       naf.entity.ts
    │       sirene-entreprise.entity.ts
    ├───schemas
    │       naf.schema.ts
```

```
│       sirene-entreprise.schema.ts
└─────services
      naf.service.ts
      sirene-entreprise.service.ts
```

L’intégralité des documents à la racine sont des documents de configuration.

Le dossier ressources contient les fichiers téléchargés lors de l’initialisation de notre BDD avec l’api SRIENE et BAN. Le principe sera détaillé plus tard dans ce document.

Le dossier script contient un docker-compose pour créer une base de données mongodb sous docker en local.

Le dossier src contient l’intégralité du code, divisé en plusieurs parties :

- **API** : Cette partie englobe les différents services qui font des appels à l'API SIRENE et BAN. Elle gère également l'intégration des fichiers CSV téléchargés depuis SIRENE.

- **Entreprise** : Cette section contient les schémas d'entités et les contrôleurs nécessaires pour gérer les actions liées aux entreprises dans l'application.

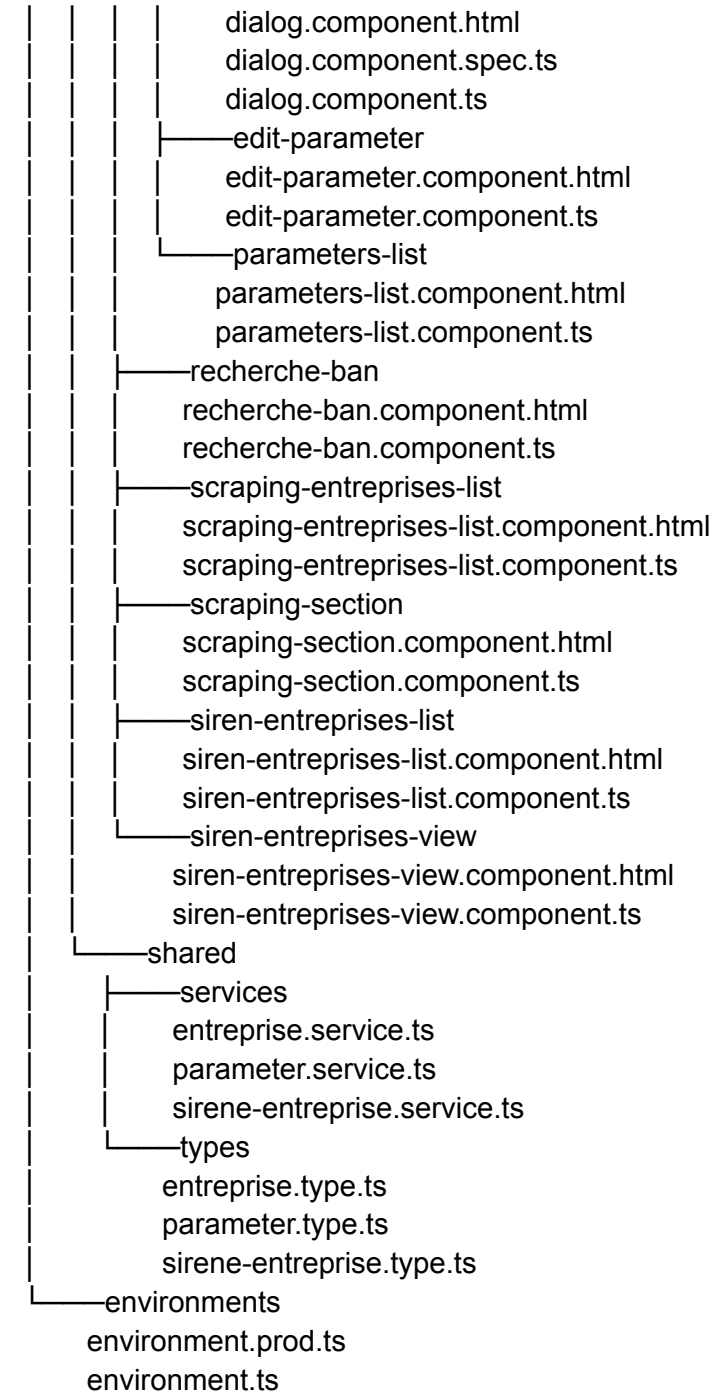
- **Paramètres** : Cette partie fonctionne de manière similaire à la partie Entreprise, mais elle est dédiée à la gestion des différents paramètres de l'application.

- **Scraping** : Cette section regroupe toutes les fonctionnalités liées au scraping des entreprises, y compris les méthodes pour récupérer des données depuis Pappers et Societe.com. Bien que cette partie ne soit plus utilisée dans l'application actuelle, elle contient également des services de scraping de sites d'avis (dans un dossier distinct nommé "review") qui ont été utilisés pour des tests. Ces services ne sont pas intégrés à l'application principale car ils ne correspondent pas aux clés de la base de données.

- **Sirene-Entreprise** : Cette partie concerne les entreprises SIRENE, détaillées plus tard dans le rapport, ainsi que des codes NAF. Elle inclut les contrôleurs et autres fichiers nécessaires au bon fonctionnement et à l'organisation du projet Nest.

Frontend

```
C:.
│   .dockerignore
│   .editorconfig
│   .gitignore
│   angular.json
│   Dockerfile
│   nginx.conf
│   package-lock.json
│   package.json
│   proxy.conf.json
│   tailwind.config.js
│   tsconfig.app.json
│   tsconfig.json
│   tsconfig.spec.json
└───src
    │   favicon.ico
    │   index.html
    │   main.ts
    │   styles.css
    ├───app
    │   │   app-routing.module.ts
    │   │   app.component.html
    │   │   app.component.ts
    │   │   app.module.ts
    │   ├───components
    │   │   ├───drawer-menu
    │   │   │       drawer-menu.component.html
    │   │   │       drawer-menu.component.spec.ts
    │   │   │       drawer-menu.component.ts
    │   │   ├───entreprise-view
    │   │   │       entreprise-view.component.html
    │   │   │       entreprise-view.component.ts
    │   │   ├───entreprises-list
    │   │   │       entreprises-list.component.html
    │   │   │       entreprises-list.component.ts
    │   │   ├───entreprises-section
    │   │   │       entreprises-section.component.html
    │   │   │       entreprises-section.component.ts
    │   │   ├───footer
    │   │   │       footer.component.html
    │   │   │       footer.component.ts
    │   │   ├───navigation-bar
    │   │   │       navigation-bar.component.html
    │   │   │       navigation-bar.component.ts
    │   │   ├───parameter
    │   │   │   ├───dialog
```



Dans la structure du projet, plusieurs fichiers de configuration sont présents à la racine, tels que ceux pour Angular, Tailwind CSS, ainsi que les fichiers nécessaires à la création de conteneurs Docker pour le serveur.

Le dossier src contient également l'intégralité du code de l'application frontend, avec main.ts comme point d'entrée à la racine de ce dossier. Dans le dossier app, on retrouve principalement le fichier définissant les différentes routes de l'application.

Les différents composants (components) correspondent aux différentes parties que nous assemblons dans notre application.

Le dossier shared regroupe les divers types correspondant aux entités de notre backend, ainsi que les services qui permettent de les consommer.

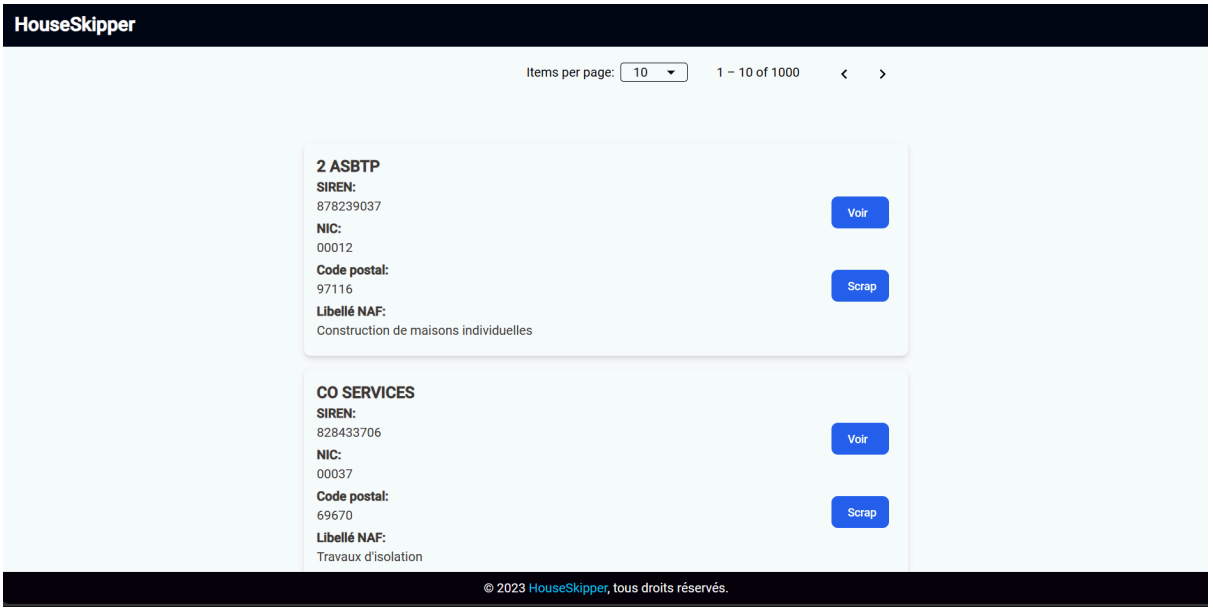
Enfin, le dossier environments contient les différentes variables d'environnement. Il convient de noter qu'elles sont push sur Git, mais à terme, il faudra les retirer si elles contiennent des variables sensibles.

Fonctionnement de l’application en local

Une fois le frontend et le backend lancé en parallèle de la base de données MongoDB.
On peut accéder à l’application (en local) en suivant ce lien : https://localhost:4200
On peut également accéder au Swagger de l’api en suivant ce lien : http://localhost:3000/api
On peut donc effectuer les différentes requêtes depuis Postman ou d’autres applications frontend.

L’application se présente principalement en 4 pages :

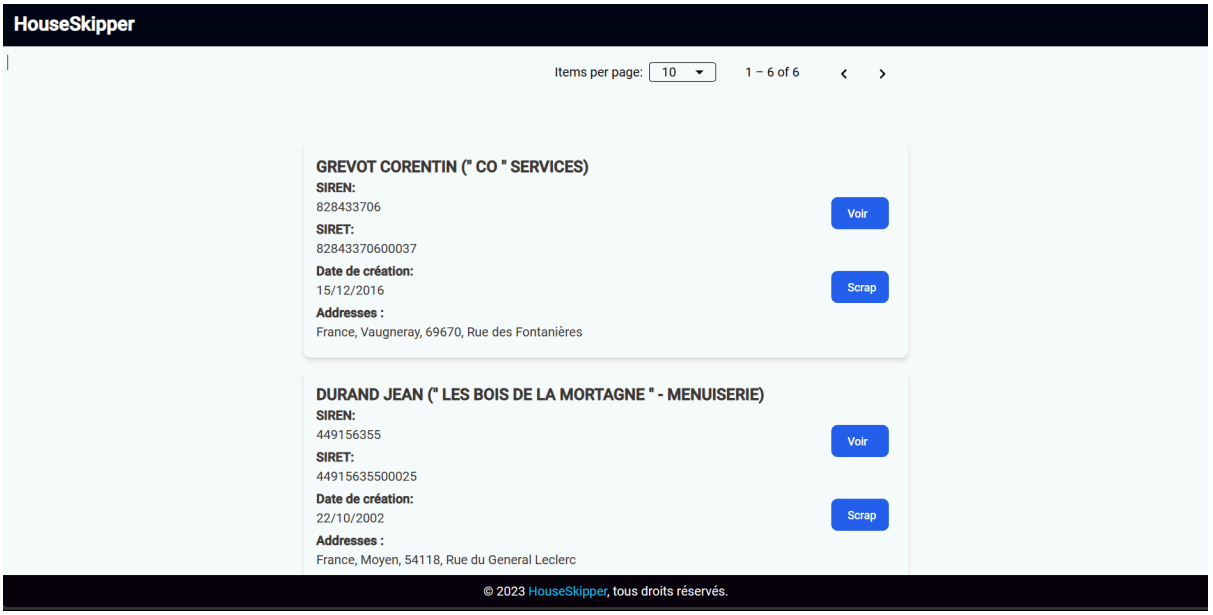
- La liste des entreprises Sirene (http://localhost:4200/sireneEntreprises) :



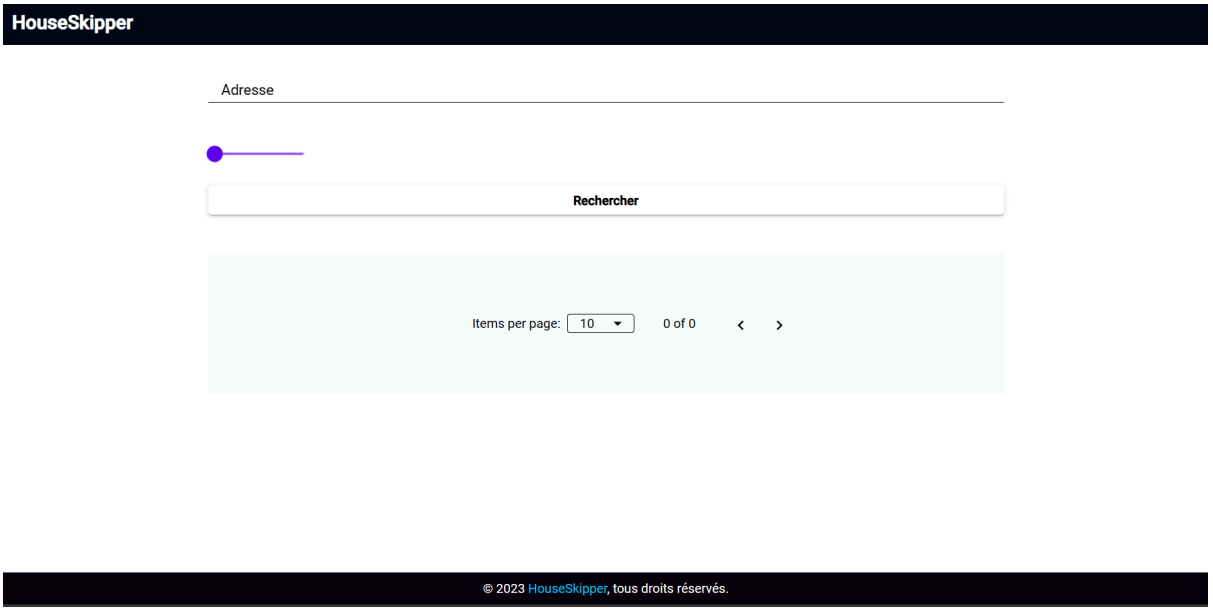
Elle présente une liste d’entreprises récupérées sur le csv de sirène. Pour que le chargement soit possible et rapide, seulement 1000 entreprises sont récupérées et affichées.

Chaque entreprise dispose d’un bouton permettant de l’afficher ou de forcer un scraping, si on clic sur “Voir” alors l’application essayera dans un premier temps de charger les données déjà présente dans la base de données, si le dernier scraping est trop vieux ou si l’on force le scraping, alors l’application backend effectuera un nouveau scraping de l’entreprise concernée.

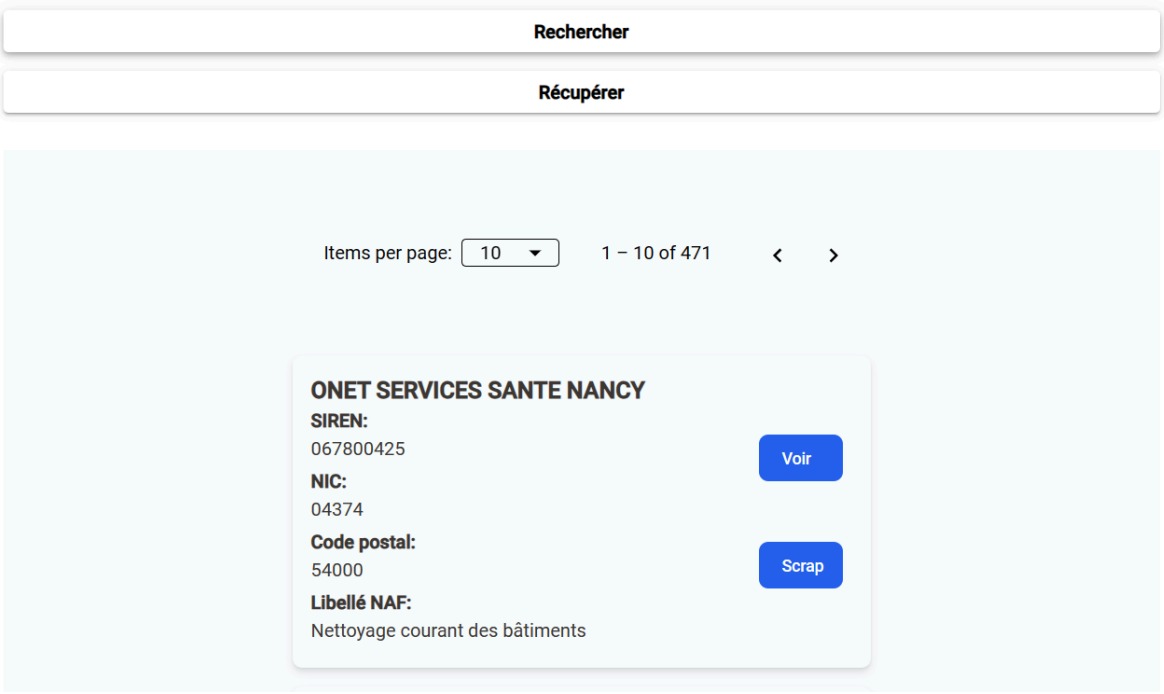
- La seconde page regroupe sous le même format la même liste, mais cette fois-ci uniquement avec les entreprises déjà scrappées (http://localhost:4200/scrapingEntreprises) :



- La troisième page permet de saisir une entreprise et une distance et de récupérer la liste des entreprises dans cette même zone (http://localhost:4200/sireneEntreprisesSearch).



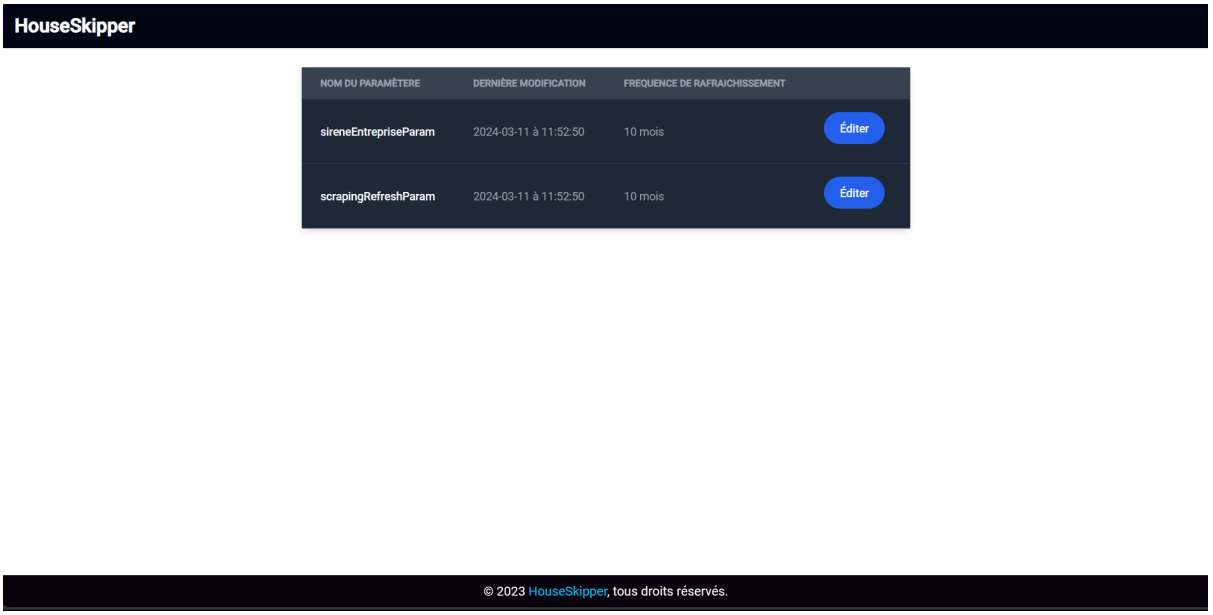
Ici les résultats pour une recherche aux alentours de Nancy :



Là encore la liste permet de scraper ou de voir les différentes entreprises, au-dessus un nouveau bouton apparaît “Récupérer”, et permet de scraper la liste des entreprises récupérées en dessous.
Ici, nous faisons fasse à un problème suite à l’utilisation du scraping, en effet :

Nous parlerons donc des différentes solutions envisageables pour régler ce problème plus bas.

- La dernière page présente les différents paramètres de scraping et permet de les régler (changer les temps) :



De plus, sur la page d'accueil, un bouton “export JSON” est disponible pour extraire l’intégralité des données sur les entreprises scrapées au format json.

Fonctionnement de l’application sur le serveur

Actuellement, le déploiement de l'application sur le serveur nécessite de copier le contenu du dépôt Git sur le serveur, suivi de l'exécution du docker-compose. Cependant, cette méthode a posé des problèmes de place, ce qui a nécessité le remplissage de la base de données à distance.

Bien que cette approche fonctionne, elle n'est pas optimale et présente des inconvénients en termes de gestion et de scalabilité. Toutefois, en raison de contraintes techniques et de contraintes de temps, nous avons dû l'adopter temporairement.

Pour améliorer le fonctionnement du serveur, plusieurs pistes à envisager sont décrites dans la section suivante.

La BDD MongoDB : http://51.68.95.251:27020/
Le Backend NestJS : http://51.68.95.251:3020/
La documentation Swagger : http://51.68.95.251:3020/api
Le frontend Angular : http://51.68.95.251:4220/

Perspectives d'améliorations à envisager pour l'application

Comme discuté précédemment, plusieurs pistes d'amélioration doivent être envisagées. Tout d'abord, il serait judicieux d'explorer des solutions permettant de contourner les limitations persistantes du scraping, telles que l'utilisation de proxies ou simplement l'intégration d'APIs lorsque disponibles.

En outre, il est crucial d'intensifier les tests au sein de l'application et d'optimiser certains aspects du code pour garantir sa fiabilité et ses performances.

En ce qui concerne la mise en place d'une chaîne de déploiement automatique, un premier pas a été franchi avec la création d'images et de conteneurs. Cependant, pour une gestion plus aisée des mises à jour depuis le serveur, il serait préférable de les héberger directement sur Docker Hub ou une autre plateforme similaire, et d'adapter en conséquence le docker-compose.

De plus, il est impératif de résoudre le problème de taille du serveur. Bien qu'un nouveau disque soit disponible, il n'est actuellement pas utilisé, ce qui entrave le fonctionnement de l'application (notamment Puppeteer qui ne peut pas créer son cache, empêchant ainsi le scraping lorsque le serveur est saturé).

Ces améliorations permettront d'optimiser le fonctionnement de l'application, d'améliorer sa stabilité et sa scalabilité, tout en assurant une gestion plus efficace des ressources serveur.