



# Pré-requis

## Cas général

Il est nécessaire d'installer en amont :

- [Docker](#) ;
- [NodeJS](#)

Une fois ces logiciels installés, merci d'exécuter ces commandes afin d'installer les modules nécessaires :

```
cd front && npm install  
cd ../server && npm install
```

## Base de données

Nous utilisons MongoDB comme serveur de base de données. Pour des raisons pratiques, nous avons décidé d'utiliser un serveur cloud (afin de partager plus facilement les informations de celle-ci d'une machine à une autre).

Pour faire de même, il vous suffira de créer un compte sur [MongoDB Cloud](#) et de créer un projet. Vous pourrez ensuite dans la section **Databases** récupérer les informations de connexion à votre cluster en cliquant sur le bouton **Connect** puis **Connect your application** . Vous trouverez ensuite l'URL de connexion à remplacer dans la variable d'environnement **MONGO\_URL** du fichier **docker-compose.yml** .

## Mode développeur

Pour pouvoir modifier le projet, il est nécessaire d'installer en supplément :

- [CLI d'Angular](#)



# Utilisation

## Utilisation normale

Pour lancer l'application, il vous suffit de construire les images Docker et de lancer le container :

```
docker compose up --build
```

Une fois les images construites une première fois, vous pourrez simplement lancer le container les fois suivantes :

```
docker compose up
```

Une fois lancé, l'interface utilisateur sera accessible à l'adresse <http://localhost> tandis que l'API sera accessible à l'adresse <http://localhost:8080>.

## Mode développeur

Le passage par Docker peut compliquer les tâches de développement des entités. En effet, il faudrait reconstruire les images à chaque modification, ce qui peut prendre du temps. Pour éviter cela, vous pouvez lancer les entités une par une :

## Lancement du backend

```
cd ./server  
npm start
```

Une fois lancée, l'API sera accessible à l'adresse <http://localhost:8080>.



## Lancement du frontend

```
cd ./front
ng serve --open
```

### Attention

La variable d'environnement `MONGO_URL` ne sera pas prise en compte dans ce cas ! Il est nécessaire de recopier l'URL d'accès à la base de données dans le fichier `./server/config.js`. Ce fichier de configuration sera utilisé dans le cas d'un lancement hors-Docker.

Une fois lancée, l'interface utilisateur sera accessible à l'adresse <http://localhost:4200>.

## Visualisation manuelle

Des fichiers déjà calculés permettent de tester l'application. Vous trouverez dans le dossier `test_files` ces différents fichiers.

Dans le tableau ci-dessous, vous trouverez les coordonnées GPS de Pékin et de San Francisco. Choisissez une des deux villes et renseignez les coordonnées dans les champs correspondants.

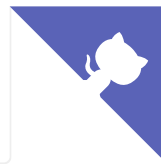
Ville	Latitude	Longitude
Pékin	39.928386	116.392004
San Francisco	37.770366	-122.442536

Choisissez ensuite le fichier `Grid` et le fichier `Color` de la ville voulue. Vous pouvez ensuite cliquer sur le bouton `Afficher` !

## Visualisation automatique

### Remarque

Il est nécessaire d'être connecté pour utiliser cette fonctionnalité.



Il est possible de sauvegarder en base de données les données d'une ville afin de faciliter leur visualisation (voir les chapitres [Insertion manuelle](#) et [automatique](#)). La visualtion automatique permet d'afficher une ville sauvegardée.

Il vous suffit de choisir les différents champs et de cliquer sur **Afficher**.

## Insertion manuelle


### Remarque

Il est nécessaire d'être connecté pour utiliser cette fonctionnalité.

Il est possible de sauvegarder en base de données les données d'une ville afin de faciliter leur visualisation. La [visualisation automatique](#) permet ensuite d'afficher une ville sauvegardée.

Il vous suffit de remplir les champs du formulaire. Les champs sont expliqués ci-dessous :

Nom	Description	Exemple
Ville	Nom de la ville	Pekin
Latitude		
Longitude		
Période	Combien d'heures la mesure affiche-t-elle ?	24
Granularité	Combien de mètres un carré couvre-t-il ?	0.06
Seuil de voitures	Combien de voitures sont nécessaires pour considérer la donnée viable ?	0.021
Division du temps	Timestamp de la première mesure	
Division du temps	Timestamp de la dernière mesure	
Direction	Avec ou Sans la direction (une flèche peut avoir deux sens)	

Fichier Grid			
Fichier Color			

## Insertion automatique

### Remarque

Il est nécessaire d'être connecté pour utiliser cette fonctionnalité.

L'insertion manuelle étant longue à renseigner, il est possible d'insérer automatiquement une ville en base de données. Pour ce fonctionnement, les fichiers **Grid** et **Color** doivent être nommés sous le format suivant :  
`type_ville_latitude_longitude_période_granularite_sueil_time1_time2_direct`

Le choix du séparateur (ici `_`) est laissé libre et peut être renseigné dans le formulaire.

Il est également possible d'importer plusieurs fichiers en cochant la case correspondante.

## Environnement

### Architecture

L'application est axée autour de trois services : le serveur API (en NodeJS / Express), l'interface utilisateur (en Angular) et la base de données (MongoDB).

 Architecture

Service	Port
Backend (node)	8080
Frontend (angular)	4200
Frontend (docker)	80
Base de données (mongodb)	Service cloud

# Environnement logiciel



## API

Nom du logiciel	Version	Description
Node	14.18.0	Serveur web
Body Parser	1.18.3	Parsing du corps des requêtes
Express	4.16.4	Gestion des routes et services web
GridFS-Stream	1.1.1	Permet de lire des fichiers GridFS depuis Mongo
JsonWebToken	8.4.0	Gestion des tokens JWT
Mongoose	5.4.3	Gestion de la connexion à MongoDB
ShellJS	0.8.3	Utilisation de commandes Shell en JS
UUID	8.3.2	Génération d'identifiants utilisateurs uniques

## Frontend

Nom du logiciel	Version	Description
@angular	12.2.0	Angular
Leaflet	1.7.1	API de cartographie
RXJS	6.6.0	Observables
TsLib	2.3.0	TypeScript helpers

## Authentification

### Connexion

Les utilisateurs doivent pouvoir se connecter pour utiliser certaines fonctionnalités de l'application. Pour se connecter, un utilisateur doit cliquer sur le bouton de connexion. Il renseigne ensuite son adresse e-mail et son

mot de passe. Une fois connecté, il est redirigé sur l'écran de visualisation manuelle.



Lors de la connexion, un token JWT sera généré et permettra de sécuriser les échanges avec l'API.

Endpoint	Méthode	Succès	Erreur
<code>/login</code>	POST	<code>{success:true, token: TOKEN}</code>	<code>{success:false, error: ERROR }</code>

Le loadout de la requête API est :

json

```
{  
  "email": "<email>",  
  "password": "<password>"  
}
```

Le statut de la connexion est sauvegardé par le frontend dans le stockage local du navigateur. Cette variable contiendra le token JWT renvoyé en cas de succès.

## Inscription

Les utilisateurs doivent pouvoir s'inscrire. Pour s'inscrire, un utilisateur doit cliquer sur le bouton d'inscription et remplir le formulaire avec son adresse e-mail et son mot de passe. Il pourra ensuite se connecter.

Endpoint	Méthode	Succès	Erreur
<code>/register</code>	POST	<code>{success:true}</code>	<code>{success:false, error: ERROR }</code>

Le loadout de la requête API est :

json

```
{  
  "email": "<email>",  
}
```

```
"password" : "<password>"  
}
```



## Déconnexion

Les utilisateurs doivent pouvoir se déconnecter de l'application. Lorsque l'utilisateur cliquera sur le bouton de déconnexion, le stockage local contenant le token JWT sera supprimé et l'application reviendra sur la page de visualisation manuelle.

## Visualisation

### Introduction

Ce cas d'usage permet d'afficher sur une carte les résultats de l'analyse de trajectoires dans le but d'en faciliter la lecture. Cette étape affiche ainsi une matrice de carrés (le côté du carré peut être défini). Ces carrés seront colorés en fonction du résultat de l'analyse.

Aussi, il est possible de cliquer sur un carré pour ouvrir une pop-up contenant quelques informations complémentaires (position GPS, signification de la couleur).

Dans un cas particulier (avec direction), il est possible d'ajouter aux carrés une ou plusieurs flèches indiquant l'évolution du trafic dans une ou plusieurs direction(s).

### Visualisation manuelle

Les utilisateurs peuvent visualiser les résultats de manière manuelle sur la carte. Pour se faire, ils doivent renseigner la latitude et la longitude centrale, un fichier Color et un fichier Grid.



# Visualisation automatique



Les utilisateurs peuvent charger depuis la base de données et visualiser les résultats de manière automatique sur la carte. Pour se faire, ils doivent choisir dans une arborescence les différents paramètres proposés.

Les paramètres sont récupérés grâce à une requête à l'API :

Endpoint	Méthode	Succès
<code>/getCities</code>	POST	<code>{cities:CITIES}</code>

Le loadout de la requête est le suivant :

```
json
{
  "token": "<token>"
}
```

Nous affichons ensuite le premier paramètre, puis à chaque nouveau choix, nous filtrons les choix suivants par rapport à ceux déjà faits.

## Affichage sur la carte

Dans les deux cas précédents, en envoyant le formulaire, le front-end se charge de transmettre au composant de la carte les différentes données via le service de partage, grâce à cette ligne :

```
js
self.sharedService.nextMessage({lat: f.value.lat, lon: f
```

Le composant de la carte gère ensuite de l'affichage (voir fonctionnement dans [Map](#)).

## Insertion

# Introduction



Un utilisateur connecté peut ajouter un résultat en base de données. Ce résultat pourra ensuite être chargé plus facilement via la [visualisation automatique](#).

## Insertion manuelle

Les utilisateurs peuvent insérer en base de données un résultat de manière manuelle. Pour ce faire, ils doivent renseigner tous les paramètres du résultat (position GPS, période, granularité...) grâce à un formulaire puis charger les fichiers **Grid** et **Color** correspondants. Une fois le formulaire rempli, le front-end se charge de lire les fichiers transmis et de faire une requête à l'API pour ajouter l'entrée en base de données.

## Insertion automatique

Les utilisateurs peuvent insérer en base de données un résultat de manière automatique. Le fonctionnement est sensiblement identique au paragraphe précédent. La différence se situe au chargement des paramètres du résultat qui se fait de manière autonome. L'utilisateur doit à la place renseigner un séparateur (par ex : **\_**) ; l'application lira ensuite les paramètres dans le nom du fichier **Grid**. Une fois le formulaire rempli, le front-end se charge de lire les fichiers transmis et de faire une requête à l'API pour ajouter l'entrée en base de données.

## Requête API

Dans les deux cas, la requête à l'API est la suivante :

Endpoint	Méthode	Succès	Erreur
<b>/insertFiles</b>	POST	<b>{success:RESULT}</b>	<b>{success:fa error: ERROR}</b>

Le loadout de la requête est le suivant :



```
{  
  "data": "<data>",  
  "token": "<token>"  
}
```

# Fichiers générés

## Introduction

Les fichiers générés contiennent toutes les informations permettant l'affichage des résultats obtenus sur la carte. Ils contiennent des coordonnées et les types des résultats (permettant de connaître la couleur à afficher). Ces fichiers sont générés via des scripts `bash` exécutant un fichier `jar`. Aucune documentation sur la génération des fichiers n'a été trouvée à la reprise du projet en 2021.

Des fichiers permettant de tester l'application sont contenus dans le dossier `/test_files`.

## Grid

Un fichier `Grid` possède des lignes contenant les différentes informations d'affichage de chaque carré à afficher. Chaque ligne est au format suivant :

```
<id>|<coord:1>|<coord:2>|<coord:3>|<coord:4>|
```

Les `<coord>` sont au format `<lat>,<lon>`. Ils représentent les extrémités de chacun des carrés. L' `<id>` est unique et est également référencé dans les fichiers `Color` (voir ci-dessous).

## Color

Un fichier **Color** possède des lignes contenant les couleurs des carrés à afficher. Chaque ligne est au format suivant :



<id> - <color>

Les différentes couleurs disponibles sont :

Nom de la couleur	Signification	Couleur
EMERGING	Le trafic augmente	
DECREASING	Le trafic diminue	
LATENT		
JUMPING		
LOST	La donnée n'est plus disponible	
DEFAULT	Couleur par défaut	

## Carte

### Présentation des fonctions

La carte permet de représenter le trafic routier en lisant les fichiers proposés par l'utilisateur ou grâce aux informations stockées en base de données. Les fonctions utilisées pour lire les fichiers et afficher le trafic sont les suivantes :

Nom fonction	Prototype	Descript
ngAfterViewInit	ngAfterViewInit()	Permet des modifications après son ir notamment latitude ou changent ou l'utilisateur trafic routie
initMap	initMap()	Initialise la demandant à l'utilisateur afficher sa p

		ajoute une ligne copyright sur la carte
drawLine	drawLine(centrelat: number, centrelong: number, calculatedLat: any, calculatedLong: any, couleur: any, direction: string, descriptionColor: string, numberRecColor: string, tabLatitudeRecSW: any, tabLatitudeRecNE: any)	Ajoute une ligne de coordonnées et lui attribue une couleur contenant des informations sur sa direction (Emerging, etc)
clearMap	clearMap()	Appelle les fonctions supprimant des formes sur la map
clearRectangles	clearRectangles()	Supprime tous les rectangles de la map
clearArrow	clearArrow()	Supprime toutes les flèches dessinées sur la map
clearColor	clearColor()	Supprime tous les rectangles de couleur créés par la fonction areaColorTraitement
areaTraitement	areaTraitement(lines: string[], _callback: function)	Crée des zones de couleur à partir d'une fonction de traitement des informations dans le tableau argument
areaColorTraitement	areaColorTraitement(lines: string[], cb: function)	Personnalise les couleurs des zones à partir d'une fonction de traitement des informations dans le tableau argument
areaTraitementSeq	areaTraitementSeq(lines: string[], _callback: function)	Fonctionne de la même manière que areaTraitement mais n'assigne pas de couleur aux zones créées
areaColorTraitementSeq	areaColorTraitementSeq(lines: string[], cb: function)	Appelle la fonction drawLine pour dessiner des flèches dans les zones créées par areaTraitement



		de nouvelle couleur si c lignes du ta argument n comportent directions
--	--	---

