

Supmap : Documentation Technique

Ce document est un merge de la documentation technique de chaque microservice.

Sommaire

- [Supmap : Documentation Technique](#)
 - [Sommaire](#)
 - [Gateway](#)
 - [Fonctionnalités](#)
 - [Architecture](#)
 - [Configuration](#)
 - [Démarrage](#)
 - [Développement](#)
 - [Users](#)
 - [Présentation](#)
 - [Architecture](#)
 - [Prérequis et installation](#)
 - [Configuration](#)
 - [Swagger](#)
 - [Authentification](#)
 - [Migrations de base de données](#)
 - [Endpoints](#)
 - [GIS \(Geographic Information System\)](#)
 - [1. Introduction](#)
 - [2. Architecture générale](#)
 - [3. Organisation du projet et Structure des dossiers](#)
 - [4. Détail des services internes](#)
 - [5. Endpoints HTTP exposés](#)
 - [6. Structures & interfaces importantes](#)
 - [7. Stack trace & appels typiques](#)
 - [8. Configuration et build](#)
 - [Incidents](#)
 - [Présentation](#)
 - [Architecture](#)
 - [Prérequis et installation](#)
 - [Configuration](#)
 - [Swagger](#)
 - [Authentification](#)
 - [Migrations de base de données](#)
 - [Auto-modération des incidents](#)
 - [Communication par Redis Pub/Sub](#)
 - [Gestion des transactions SQL concurrentes](#)
 - [Endpoints](#)
 - [Navigation](#)

- 1. Introduction
- 2. Architecture générale
- 3. Organisation du projet et Structure des dossiers
- 4. Détail des services internes
- 5. Endpoint HTTP exposé
- 6. Protocole & messages WebSocket
- 7. Structures & interfaces importantes
- 8. Stack trace & appels typiques
- 9. Configuration
- Application mobile (front)
 - Choix des technologies
 - Structure du projet
 - Authentification
 - Communication avec l'API
 - Structure des écrans et composants
 - Carte
 - Calcul de l'itinéraire
 - Gestion des incidents
 - Communication avec les WebSockets
 - Récupération d'itinéraire par QR Code
 - Démarrer et tester l'application
- WebApp
 - Vue d'ensemble
 - Architecture des Composants
 - Technologies Utilisées
 - Structure du Projet
 - Fonctionnalités Clés
 - Configuration et Environnement
- DevOps
 - Authentification
 - Lancement du projet (complet)
 - Variables d'environnement
 - Services externes

Gateway

Une gateway API légère et performante écrite en Go, permettant de router et de rediriger les requêtes vers différents microservices.

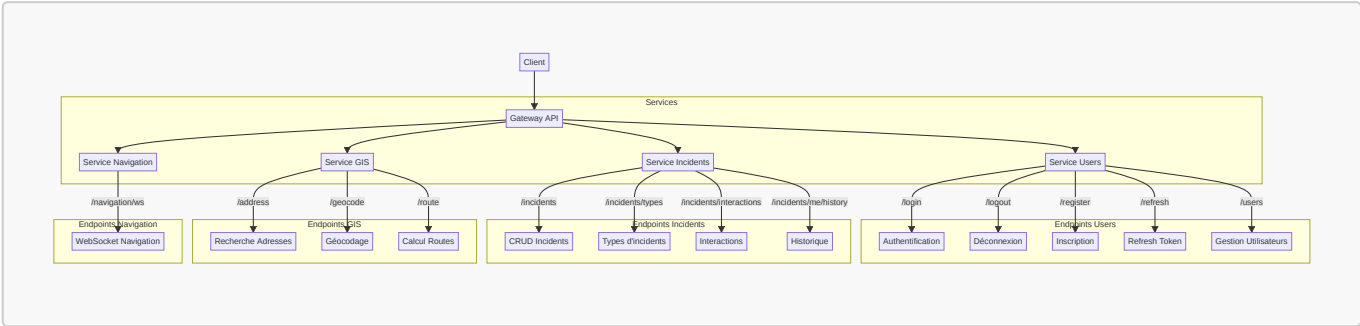
Fonctionnalités

- Reverse proxy pour plusieurs microservices :
 - Service Utilisateurs (authentification, gestion des comptes)
 - Service Incidents (gestion et suivi des incidents)
 - Service GIS (géocodage et services géographiques)
 - Service Navigation (WebSocket pour la navigation en temps réel)
- Configuration simple via variables d'environnement

- Support des WebSockets
- Gestion des routes avec réécriture d'URL

Architecture

La gateway agit comme un point d'entrée unique pour différents microservices :



Configuration

La configuration se fait via des variables d'environnement :

Variable	Description
SUPMAP_GATEWAY_PORT	Port d'écoute de la gateway
SUPMAP_USERS_HOST	Hôte du service utilisateurs
SUPMAP_USERS_PORT	Port du service utilisateurs
SUPMAP_INCIDENTS_HOST	Hôte du service incidents
SUPMAP_INCIDENTS_PORT	Port du service incidents
SUPMAP_GIS_HOST	Hôte du service GIS
SUPMAP_GIS_PORT	Port du service GIS
SUPMAP_NAVIGATION_HOST	Hôte du service navigation
SUPMAP_NAVIGATION_PORT	Port du service navigation

Points d'accès (Endpoints)

- Service Utilisateurs
 - /login - Authentification
 - /logout - Déconnexion
 - /register - Création de compte
 - /refresh - Rafraîchissement du token
 - /users - Gestion des utilisateurs
- Service Incidents
 - /incidents - CRUD des incidents
 - /incidents/types - Types d'incidents
 - /incidents/interactions - Interactions sur les incidents
 - /incidents/me/history - Historique personnel

- Service GIS
 - /address - Recherche d'adresses
 - /geocode - Géocodage
 - /route - Calcul d'itinéraires
- Service Navigation
 - /navigation/ws - WebSocket pour la navigation en temps réel

Démarrage

Configurez les variables d'environnement nécessaires Lancez l'application :

```
go run .
```

Développement

Le projet utilise :

- net/http/httputil pour le reverse proxy
- github.com/caarlos0/env/v11 pour la gestion de la configuration
- Go modules pour la gestion des dépendances

Users

Microservice de gestion des données utilisateurs pour Supmap

Présentation

supmap-users est un microservice écrit en Go destiné à fournir l'ensemble des fonctionnalités directement liées à un utilisateur. Il intègre une gestion de compte complète et d'authentification. Ce service est complètement stateless et peut être déployé selon une scalabilité verticale à condition de fournir les mêmes paramètres d'environnement.

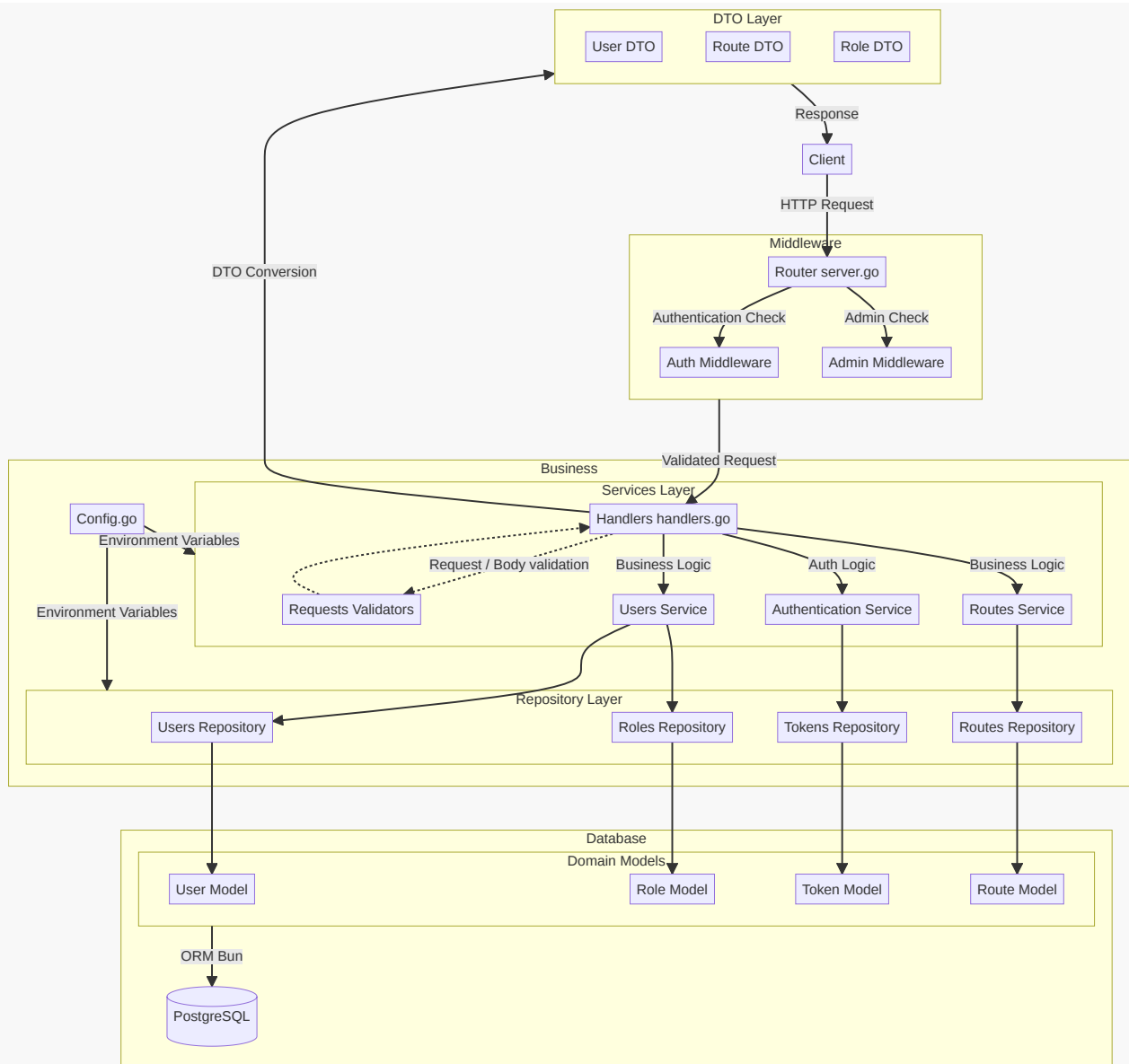
Différents endpoints sont exposés :

- Authentification (Register, Login, Refresh etc...)
- Modification du compte
- Gestion des itinéraires enregistrés

Certaines routes sont publiques, d'autres nécessitent une authentification et d'autres sont réservés aux utilisateurs authentifiés comme administrateur.

Architecture

Ce service implémente une architecture par couche avec des controllers (handlers), des services et des repositories.



```

supmap-users/
├── cmd/
│   └── api/
│       └── main.go           # Point d'entrée du microservice
├── internal/
│   ├── api/
│   │   ├── handlers.go      # Gestionnaires de requêtes HTTP
│   │   ├── server.go        # Configuration du serveur HTTP et routes
│   │   ├── middlewares.go   # Intercepteurs de requête
│   │   └── validations/
│   │       └── ...          # Structures de validation
│   ├── config/
│   │   └── config.go        # Configuration des variables d'environnement
│   ├── models/
│   │   ├── dto/             # DTOs permettant d'exposer les données
│   │   └── ...              # Structures de données pour l'ORM Bun
│   └── repository/          # Repository implémentant les requêtes SQL
└── avec l'ORM Bun
    └── ...

```

```
|   └─ services/           # Services implémentant les fonctionnalités
métier du service
|       └─ ...
└─ docs/                   # Documentation Swagger auto implémentée avec
Swagger
|   └─ ...
└─ Dockerfile              # Image Docker du microservice
└─ go.mod                  # Dépendances Go
└─ go.sum                  # Checksums des dépendances (auto généré)
└─ README.md               # Documentation du projet
```

Prérequis et installation

- Go 1.24
- Base de données postgres (conteneurisée ou non)

Démarrage rapide

```
# Cloner le repo
git clone https://github.com/4PROJ-Le-projet-d-une-vie/supmap-users.git
cd supmap-users

# Démarrer le service (nécessite les variables d'environnement, voir ci-
dessous)
go run ./cmd/api
```

Avec Docker

```
docker pull ghcr.io/4proj-le-projet-d-une-vie/supmap-users:latest
docker run --env-file .env -p 8080:80 supmap-users
```

NB: Nécessite une d'être authentifié pour accéder aux artefacts dans l'organisation Github du projet.

Authentification au registre GHCR

Pour pull l'image, il faut être authentifié par docker login.

- Générer un Personal Access Token sur GitHub :
 - Se rendre sur <https://github.com/settings/tokens>
 - Cliquer sur "Generate new token"
 - Cocher au minimum la permission read:packages
 - Copier le token
- Connecter Docker à GHCR avec le token :

```
echo 'YOUR_GITHUB_TOKEN' | docker login ghcr.io -u YOUR_GITHUB_USERNAME --password-stdin
```

Configuration

La configuration se fait via des variables d'environnement ou un fichier `.env` :

Variable	Description
<code>ENV</code>	Définit l'environnement dans lequel est exécuté le programme (par défaut production)
<code>DB_URL</code>	URL complète vers la base de donnée
<code>PORT</code>	Port sur lequel écoute le service pour recevoir les requêtes
<code>JWT_SECRET</code>	Secret permettant de vérifier l'authenticité d'un token JWT pour l'authentification

Ces variables sont chargés depuis le fichier `config.go`, à l'aide de la librairie `caarlos0/env`.

Swagger

Chaque handler de ce service comprend des commentaires `Swaggo` pour créer dynamiquement une page Swagger-ui. Exécutez les commandes suivantes depuis la racine du projet pour générer la documentation :

```
# Installez l'interpréteur de commande Swag
go install github.com/swaggo/swag/cmd/swag@latest

# Générez la documentation
swag init -g cmd/api/main.go
```

Maintenant, vous pouvez accéder à l'URL `http://localhost:8080/swagger/index.html` décrivant la structure attendue pour chaque endpoint de l'application

NB: La documentation n'inclut pas les endpoints `/internal` destinés à une utilisation exclusivement interne

Authentification

Cette API utilise un système d'authentification basé sur JWT (JSON Web Tokens) avec un access token de courte durée et un refresh token de longue durée.

NB: La description détaillée des endpoints est disponible dans la section suivante (Endpoints)

1. L'utilisateur peut s'enregistrer via l'endpoint `/register` en fournissant les informations suivantes dans le corps de la requête.
2. L'utilisateur peut se connecter via l'endpoint `/login` en utilisant **soit son email, soit son handle**, accompagné de son mot de passe
 - La réponse contient deux tokens, un **access_token valable 24h** et un **refresh_token valable 1 an**

- L'`access_token` est à mettre dans le header `Authorization` pour être autorisé à appeler des endpoints sécurisés

```
Authorization: Bearer <access_token>
```

3. Quand l'`access_token` expire, l'utilisateur peut obtenir un nouveau token sans se reconnecter via l'endpoint `/refresh`
4. L'utilisateur peut se déconnecter via l'endpoint `/logout`
 - Cela invalide tous les tokens actifs (access + refresh).
 - Un nouveau `refresh_token` sera généré au prochain login.

Migrations de base de données

Les migrations permettent de versionner la structure de la base de données et de suivre son évolution au fil du temps. Elles garantissent que tous les environnements (développement, production, etc.) partagent le même schéma de base de données.

Ce projet utilise `Goose` pour gérer les migrations SQL. Les fichiers de migration sont stockés dans le dossier `migrations/changelog/` et sont embarqués dans le binaire grâce à la directive `//go:embed` dans `migrate.go`.

Création d'une migration

Pour créer une nouvelle migration, installez d'abord le CLI `Goose` :

```
go install github.com/pressly/goose/v3/cmd/goose@latest
```

Puis créez une nouvelle migration (la commande se base sur les variables du fichier `.env`) :

```
# Crée une migration vide
goose -dir migrations/changelog create nom_de_la_migration sql

# La commande génère un fichier horodaté dans migrations/changelog/
# Exemple: 20240315143211_nom_de_la_migration.sql
```

Exécution des migrations

Les migrations sont exécutées automatiquement au démarrage du service via le package `migrations` :

```
// Dans main.go
if err := migrations.Migrate("pgx", conf.DbUrl, logger); err != nil {
    logger.Error("migration failed", "err", err)
}
```


Le package migrations utilise embed.FS pour embarquer les fichiers SQL dans le binaire :

```
//go:embed changelog/*.sql
var changelog embed.FS
// Connexion à la base de données
goose.SetBaseFS(changelog)
```

Endpoints

Les endpoints ci-dessous sont présentés selon l'ordre dans lequel ils sont définis dans [server.go](#)

GET /users

Cet endpoint permet à un utilisateur authentifié en tant qu'administrateur, d'accéder à la liste de tous les utilisateurs existants.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- L'utilisateur doit avoir le rôle d'administrateur (sinon code http 403)

Paramètres / Corp de requête

Aucun paramètre ni corp de requête n'est requis pour cette requête

Réponse

```
[
  {
    "id": 0,
    "email": "string",
    "handle": "string",
    "auth_provider": "string",
    "profile_picture": "string",
    "role": {
      "id": 0,
      "name": "string"
    },
    "created_at": "string",
    "updated_at": "string"
  },
  ...
]
```

NB: La clé `auth_provider` est utilisée pour savoir si l'utilisateur utilise un compte local ou distant avec de l'OAuth. Elle n'a pas été implémentée dans le projet donc sa valeur est toujours définie à "local"

Trace

```
mux.Handle("GET /users", s.AuthMiddleware()(s.AdminMiddleware()
(s.GetUsers()))))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)      # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool      # Vérifie que la session de l'utilisateur est
valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error)   # Récupère le refresh_token de l'utilisateur
|> func (s *Server) AdminMiddleware() func(http.Handler) http.Handler
# Vérifie que l'utilisateur authentifié soit un administrateur
|> func (s *Server) GetUsers() http.HandlerFunc
# Handler HTTP
|> func (s *Service) GetAllUsers(ctx context.Context) ([]models.User,
error)      # Service
|   |> func (u *Users) FindAll(ctx context.Context) ([]models.User,
error)      # Repository
|> func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
|> mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error      # Ecriture de la réponse avec une fonction
générique
```

GET /users/{id}

Cet endpoint permet à un utilisateur authentifié en tant qu'administrateur d'accéder aux informations détaillées d'un utilisateur spécifique.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- L'utilisateur doit avoir le role d'administrateur (sinon code http 403)

Paramètres / Corps de requête

Paramètre	Type	Description
id	int64	Identifiant de l'utilisateur

Réponse

```
{
  "id": 0,
  "email": "string",
  "handle": "string",
  "auth_provider": "string",
  "profile_picture": "string",
  "role": {
    "id": 0,
    "name": "string"
  },
  "created_at": "string",
  "updated_at": "string"
}
```

Trace

```
mux.Handle("GET /users/{id}", s.AuthMiddleware()(s.AdminMiddleware()
(s.GetUserById())))
└─> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   └─> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)          # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   └─> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)          # Repository
|   └─> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool            # Vérifie que la session de l'utilisateur
est valide
|   └─> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error)        # Récupère le refresh_token de l'utilisateur
└─> func (s *Server) AdminMiddleware() func(http.Handler) http.Handler
# Vérifie que l'utilisateur authentifié soit un administrateur
└─> func (s *Server) GetUserById() http.HandlerFunc
# Handler HTTP
└─> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)          # Service
|   └─> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)          # Repository
└─> func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
└─> mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error    # Ecriture de la réponse avec une
fonction générique
```

GET /users/me

Cet endpoint permet à un utilisateur authentifié d'accéder à ses propres informations.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

Aucun paramètre ni corps de requête n'est requis pour cette requête. Les données de l'utilisateur sont récupérée avec son `access_token`

Réponse

```
{
  "id": 0,
  "email": "string",
  "handle": "string",
  "auth_provider": "string",
  "profile_picture": "string",
  "role": {
    "id": 0,
    "name": "string"
  },
  "created_at": "string",
  "updated_at": "string"
}
```

Trace

```
mux.Handle("GET /users/me", s.AuthMiddleware()(s.GetMe()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)          # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)          # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool             # Vérifie que la session de l'utilisateur
est valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error)         # Récupère le refresh_token de l'utilisateur
|   |   |> func (s *Server) GetMe() http.HandlerFunc
# Handler HTTP
|   |   |> func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
|   |   |> mathdeordr.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error    # Ecriture de la réponse avec une
fonction générique
```

POST /login

Cet endpoint permet à un utilisateur de s'authentifier en utilisant soit son email, soit son handle avec son mot de passe. En cas de succès, il reçoit un access token et un refresh token.

Authentification / Autorisations

Aucune authentification n'est requise pour cet endpoint.

Paramètres / Corps de requête

```
{
  "email": "string",
  "handle": "string",
  "password": "string"
}
```

Règles de validation :

- email : Optionnel si handle fourni. Doit être un email valide
- handle : Optionnel si email fourni. Doit commencer par '@'
- password : Requis

Réponse

```
{
  "access_token": "string",
  "refresh_token": "string"
}
```

Trace

```
mux.Handle("POST /login", s.Login())
↳ func (s *Server) Login() http.HandlerFunc
# Handler HTTP
  ↳ func (s *Service) Login(ctx context.Context, email, handle *string,
password string)          # Service d'authentification
    |   ↳ func (u *Users) FindByEmail(ctx context.Context, email string)
(*models.User, error)      # Repository - recherche par email
    |   ↳ func (u *Users) FindByHandle(ctx context.Context, handle
string) (*models.User, error) # Repository - recherche par handle
    ↳ func (s *Service) Authenticate(ctx context.Context, user
*models.User)              # Génération des tokens
    |   ↳ func (t *Tokens) Insert(ctx context.Context, token
*models.Token) error        # Sauvegarde du refresh token
    ↳ mathdeordr.handler/func Encode[T any](v T, status int, w
```

```
http.ResponseWriter) error          # Ecriture de la réponse avec une
fonction générique
```

POST /register

Cet endpoint permet à un utilisateur de créer un nouveau compte. En cas de succès, il reçoit les informations de son compte ainsi qu'un access token et un refresh token pour être directement authentifié.

Authentification / Autorisations

Aucune authentification n'est requise pour cet endpoint.

Paramètres / Corps de requête

```
{
  "email": "string",
  "handle": "string",
  "password": "string",
  "profile_picture": "string"
}
```

Règles de validation :

- email : Requis, doit être un email valide
- handle : Requis, minimum 3 caractères, ne doit pas commencer par '@' (il sera ajouté automatiquement)
- password : Requis, minimum 8 caractères
- profile_picture : Optionnel, doit être une URL valide

Réponse

```
{
  "user": {
    "id": 0,
    "email": "string",
    "handle": "string",
    "auth_provider": "string",
    "profile_picture": "string",
    "role": {
      "id": 0,
      "name": "string"
    },
    "created_at": "string",
    "updated_at": "string"
  },
  "tokens": {
    "access_token": "string",
```

```

    "refresh_token": "string"
  }
}

```

Trace

```

mux.Handle("POST /register", s.Register())
↳ func (s *Server) Register() http.HandlerFunc
# Handler HTTP
|↳ func (s *Service) CreateUser(ctx context.Context, body
validations.CreateUserValidator)      # Service de création
|  |↳ func (r *Roles) FindUserRole(ctx context.Context)
(*models.Role, error)                  # Repository - récupération du rôle
par défaut
|  ↳ func (u *Users) Insert(user *models.User, ctx context.Context)
error                                  # Repository - insertion du nouvel utilisateur
|↳ func (s *Service) Authenticate(ctx context.Context, user
*models.User)                          # Génération des tokens
|  ↳ func (t *Tokens) Insert(ctx context.Context, token
*models.Token) error                  # Sauvegarde du refresh token
|↳ func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error           # Ecriture de la réponse avec une
fonction générique

```

POST /refresh

Cet endpoint permet d'obtenir un nouveau access token à partir d'un refresh token valide, sans avoir à se réauthentifier avec ses identifiants.

Authentification / Autorisations

Aucune authentification n'est requise pour cet endpoint, mais un refresh token valide doit être fourni.

Paramètres / Corps de requête

```

{
  "token": "string"
}

```

Règles de validation :

- token : Requis, doit être un refresh token valide précédemment obtenu via login ou register

Réponse

```
{  
  "access_token": "string"  
}
```

Trace

```
mux.Handle("POST /refresh", s.Refresh())  
↳ func (s *Server) Refresh() http.HandlerFunc  
# Handler HTTP  
  ↳ func (s *Service) RefreshToken(ctx context.Context, refreshToken  
string) # Service de refresh  
  |   ↳ func (t *Tokens) GetUserFromRefreshToken(ctx context.Context,  
refreshToken string) # Repository - vérifie le token et récupère  
l'utilisateur  
  ↳ func (s *Service) generateAccessToken(user *models.User)  
# Génération du nouveau token  
  ↳ mathdeodrd.handler/func Encode[T any](v T, status int, w  
http.ResponseWriter) error # Ecriture de la réponse avec une fonction  
générique
```

POST /logout

Cet endpoint permet à un utilisateur authentifié d'invalidé son refresh token actuel, le déconnectant effectivement de l'application.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

```
{  
  "token": "string"  
}
```

Règles de validation :

- token : Requis, doit être le refresh token actif de l'utilisateur authentifié

Réponse

Retourne un code 204 (No Content) en cas de succès.

Trace


```

mux.Handle("POST /logout", s.AuthMiddleware()(s.Logout()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)      # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)      # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool        # Vérifie que la session de l'utilisateur est
valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le refresh_token de l'utilisateur
|> func (s *Server) Logout() http.HandlerFunc
# Handler HTTP
|> func (s *Service) Logout(ctx context.Context, user *models.User,
refreshToken string)      # Service de déconnexion
|   |> func (t *Tokens) Delete(ctx context.Context, user *models.User)
error                      # Repository - suppression du refresh token
|   |> mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error # Ecriture de la réponse avec une fonction
générique

```

POST /users

Cet endpoint permet à un administrateur de créer un nouveau compte utilisateur avec un rôle spécifique.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- L'utilisateur doit avoir le role d'administrateur (sinon code http 403)

Paramètres / Corps de requête

```

{
  "email": "string",
  "handle": "string",
  "password": "string",
  "role": "string"
}

```

Règles de validation :

- email : Requis, doit être un email valide
- handle : Requis, minimum 3 caractères, ne doit pas commencer par '@' (il sera ajouté automatiquement)
- password : Requis, minimum 8 caractères

- role : Requis, doit être un rôle existant dans la base de données ("ROLE_USER" ou "ROLE_ADMIN")

Réponse

```
{
  "id": 0,
  "email": "string",
  "handle": "string",
  "auth_provider": "string",
  "profile_picture": "string",
  "role": {
    "id": 0,
    "name": "string"
  },
  "created_at": "string",
  "updated_at": "string"
}
```

Trace

```
mux.Handle("POST /users", s.AuthMiddleware()(s.AdminMiddleware()
(s.CreateUser()))))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error) # Récupération de l'utilisateur
à partir des informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error) # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool # Vérifie que la session de
l'utilisateur est valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le refresh_token de
l'utilisateur
|> func (s *Server) AdminMiddleware() func(http.Handler) http.Handler
# Vérifie que l'utilisateur authentifié soit un administrateur
|> func (s *Server) CreateUser() http.HandlerFunc
# Handler HTTP
|> func (s *Service) CreateUserForAdmin(ctx context.Context, body
validations.AdminCreateUserValidator) # Service
|   |> func (r *Roles) FindRole(ctx context.Context, role string)
# Repository - récupération du rôle spécifié
|   |> func (u *Users) Insert(user *models.User, ctx context.Context)
error # Repository - insertion du nouvel
utilisateur
|> func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
|> mathdeordr.handler/func Encode[T any](v T, status int, w
```

```
http.ResponseWriter) error  
avec une fonction générique
```

Ecriture de la réponse

PATCH /users/me

Cet endpoint permet à un utilisateur authentifié de mettre à jour ses propres informations. Retourne les informations mises à jour ainsi que de nouveaux tokens.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

```
{  
  "email": "string",  
  "handle": "string",  
  "profile_picture": "string"  
}
```

Règles de validation :

- email : Optionnel, doit être un email valide
- handle : Optionnel, minimum 3 caractères, ne doit pas commencer par '@' (il sera ajouté automatiquement)
- profile_picture : Optionnel, doit être une URL valide ou null pour le supprimer

Réponse

```
{  
  "user": {  
    "id": 0,  
    "email": "string",  
    "handle": "string",  
    "auth_provider": "string",  
    "profile_picture": "string",  
    "role": {  
      "id": 0,  
      "name": "string"  
    },  
    "created_at": "string",  
    "updated_at": "string"  
  },  
  "tokens": {  
    "access_token": "string",  
    "refresh_token": "string"  
  }  
}
```

```
}
}
```

Trace

```
mux.Handle("PATCH /users/me", s.AuthMiddleware()(s.PatchMe()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)          # Récupération de l'utilisateur à
partir des informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)          # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool             # Vérifie que la session de
l'utilisateur est valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error)         # Récupère le refresh_token de
l'utilisateur
|   |   |> func (s *Server) PatchMe() http.HandlerFunc
# Handler HTTP
|   |   |> func (s *Service) PatchUser(ctx context.Context, id int64, body
validations.UpdateUserValidator) # Service
|   |   |> func (u *Users) Update(user *models.User, ctx context.Context)
error                           # Repository - mise à jour de
l'utilisateur
|   |   |> func (s *Service) Authenticate(ctx context.Context, user
*models.User)                   # Génération des nouveaux tokens
|   |   |> func (t *Tokens) Insert(ctx context.Context, token
*models.Token) error             # Sauvegarde du refresh token
|   |   |> func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
|   |   |> mathdeordr.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error      # Ecriture de la réponse avec
une fonction générique
```

PATCH /users/{id}

Cet endpoint permet à un administrateur de mettre à jour les informations d'un utilisateur spécifique, y compris son rôle.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- L'utilisateur doit avoir le rôle d'administrateur (sinon code http 403)

Paramètres / Corps de requête

Paramètre	Type	Description
id	int64	Identifiant de l'utilisateur

```
{
  "email": "string",
  "handle": "string",
  "password": "string",
  "profile_picture": "string",
  "role": "string"
}
```

Règles de validation :

- email : Optionnel, doit être un email valide
- handle : Optionnel, minimum 3 caractères, ne doit pas commencer par '@' (il sera ajouté automatiquement)
- profile_picture : Optionnel, doit être une URL valide ou null pour le supprimer

Réponse

```
{
  "user": {
    "id": 0,
    "email": "string",
    "handle": "string",
    "auth_provider": "string",
    "profile_picture": "string",
    "role": {
      "id": 0,
      "name": "string"
    },
    "created_at": "string",
    "updated_at": "string"
  },
  "tokens": {
    "access_token": "string",
    "refresh_token": "string"
  }
}
```

Trace

```
mux.Handle("PATCH /users/{id}", s.AuthMiddleware()(s.AdminMiddleware()
(s.PatchUser()))))
|→ func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
```

```
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error) # Récupération de
l'utilisateur à partir des informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error) # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool # Vérifie que la
session de l'utilisateur est valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le refresh_token
de l'utilisateur
|> func (s *Server) AdminMiddleware() func(http.Handler) http.Handler
# Vérifie que l'utilisateur authentifié soit un administrateur
|> func (s *Server) PatchUser() http.HandlerFunc
# Handler HTTP
|> func (s *Service) PatchUserForAdmin(ctx context.Context, id int64,
body *validations.AdminUpdateUserValidator) # Service
|   |> func (r *Roles) FindRole(ctx context.Context, role string)
# Repository - récupération du rôle si modifié
|   |> func (u *Users) Update(user *models.User, ctx context.Context)
error # Repository - mise à jour de
l'utilisateur
|> func (s *Service) Authenticate(ctx context.Context, user
*models.User) # Génération des
nouveaux tokens
|   |> func (t *Tokens) Insert(ctx context.Context, token
*models.Token) error # Sauvegarde du
refresh token
|> func UserToDTO(user *models.User) *UserDTO
# Conversion DTO
|> mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error # Ecriture de la
réponse avec une fonction générique
```

DELETE /users/{id}

Cet endpoint permet de supprimer un compte utilisateur. Un utilisateur peut supprimer son propre compte, ou un administrateur peut supprimer n'importe quel compte.

Authentication / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- L'utilisateur doit soit être l'utilisateur ciblé, soit être administrateur (sinon code http 403)

Paramètres / Corps de requête

Paramètre	Type	Description
id	int64	Identifiant de l'utilisateur

Aucun corps de requête n'est requis pour cette requête.

Réponse

Retourne un code 204 (No Content) en cas de succès.

Trace

```
mux.Handle("DELETE /users/{id}", s.AuthMiddleware()(s.DeleteUser()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)      # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)      # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool        # Vérifie que la session de l'utilisateur est
valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le refresh_token de l'utilisateur
|   |   |> func (s *Server) DeleteUser() http.HandlerFunc
# Handler HTTP qui vérifie si l'utilisateur est admin ou propriétaire
|   |   |> func (s *Service) DeleteUser(ctx context.Context, id int64) error
# Service
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)      # Repository - vérifie l'existence de
l'utilisateur
|   |   |> func (u *Users) Delete(ctx context.Context, id int64) error
# Repository - suppression de l'utilisateur
```

PATCH /users/me/update-password

Cet endpoint permet à un utilisateur authentifié de mettre à jour son mot de passe. L'ancien mot de passe doit être fourni pour des raisons de sécurité.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

```
{
  "old": "string",
  "new": "string"
}
```

Règles de validation :

- old : Requis, doit correspondre au mot de passe actuel de l'utilisateur
- new : Requis, minimum 8 caractères, doit être différent de l'ancien mot de passe

Réponse

```
{
  "id": 0,
  "email": "string",
  "handle": "string",
  "auth_provider": "string",
  "profile_picture": "string",
  "role": {
    "id": 0,
    "name": "string"
  },
  "created_at": "string",
  "updated_at": "string"
}
```

Trace

```
mux.Handle("PATCH /users/me/update-password", s.AuthMiddleware()
(s.UpdatePassword()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error) # Récupération de
l'utilisateur à partir des informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error) # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool # Vérifie que la
session de l'utilisateur est valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le
refresh_token de l'utilisateur
|> func (s *Server) UpdatePassword() http.HandlerFunc
# Handler HTTP
|> func (s *Service) UpdatePassword(ctx context.Context, user
*models.User, body validations.UpdatePasswordValidator) # Service
|   |> func (s *Service) checkPassword(password string, user
*models.User) error # Vérifie
l'ancien mot de passe
|   |   |> func (u *Users) Update(user *models.User, ctx context.Context)
error # Repository - mise à jour
du mot de passe
|> func UserToDTO(user *models.User) *UserDTO
```



```
# Conversion DTO
↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error # Ecriture de la
réponse avec une fonction générique
```

GET /users/me/routes

Cet endpoint permet à un utilisateur authentifié de récupérer la liste de tous ses itinéraires enregistrés.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

Aucun paramètre ni corps de requête n'est requis pour cette requête.

Réponse

```
[
  {
    "id": 0,
    "name": "string",
    "route": [
      {
        "lat": 0.0,
        "lon": 0.0
      }
    ],
    "created_at": "string",
    "updated_at": "string"
  },
  ...
]
```

Trace

```
mux.Handle("GET /users/me/routes", s.AuthMiddleware()(s.getUserRoutes()))
↳ func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   ↳ func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error) # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   ↳ func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error) # Repository
|   ↳ func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool # Vérifie que la session de l'utilisateur
```

```

est valide
|      ↳ func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error)      # Récupère le refresh_token de l'utilisateur
↳ func (s *Server) getUserRoutes() http.HandlerFunc
# Handler HTTP
|↳ func (s *Service) GetUserRoutes(ctx context.Context, user
*models.User)                # Service
|      ↳ func (r *Routes) GetAllOfUser(ctx context.Context, user
*models.User)                # Repository - récupération des routes
|↳ func RouteToDTO(route *models.Route) *RouteDTO
# Conversion DTO
|↳ mathdeordr.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error    # Ecriture de la réponse avec une
fonction générique

```

GET /users/me/routes/{routeId}

Cet endpoint permet à un utilisateur authentifié de récupérer un itinéraire spécifique parmi ses routes enregistrées.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

Paramètre	Type	Description
routeId	int64	Identifiant de l'itinéraire

Aucun corps de requête n'est requis pour cette requête.

Réponse

```

{
  "id": 0,
  "name": "string",
  "route": [
    {
      "lat": 0.0,
      "lon": 0.0
    },
    ...
  ],
  "created_at": "string",
  "updated_at": "string"
}

```

Trace

```

mux.Handle("GET /users/me/routes/{routeId}", s.AuthMiddleware()
(s.GetUserRoutesById()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserById(ctx context.Context, id int64)
(*models.User, error)          # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)          # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool             # Vérifie que la session de l'utilisateur
est valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error)        # Récupère le refresh_token de l'utilisateur
|   |   |> func (s *Server) GetUserRoutesById() http.HandlerFunc
# Handler HTTP
|   |   |> func (s *Service) GetUserRouteById(ctx context.Context, userId,
routeId int64)                # Service
|   |   |   |> func (r *Routes) GetRouteUserById(ctx context.Context, userId,
routeId int64)                # Repository - récupération de la route spécifique
|   |   |   |> func RouteToDTO(route *models.Route) *RouteDTO
# Conversion DTO
|   |   |   |> mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error    # Ecriture de la réponse avec une
fonction générique

```

POST /users/me/routes

Cet endpoint permet à un utilisateur authentifié de créer un nouvel itinéraire.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

```

{
  "name": "string",
  "route": [
    {
      "lat": 0.0,
      "lon": 0.0
    },
    ...
  ]
}

```

Règles de validation :

- name : Requis, nom de l'itinéraire
- route : Requis, tableau de points contenant au moins une coordonnée
- route[].lat : Requis, latitude en degrés décimaux
- route[].lon : Requis, longitude en degrés décimaux

Réponse

```
{
  "id": 0,
  "name": "string",
  "route": [
    {
      "lat": 0.0,
      "lon": 0.0
    },
    ...
  ],
  "created_at": "string",
  "updated_at": "string"
}
```

Trace

```
mux.Handle("POST /users/me/routes", s.AuthMiddleware()
(s.CreateUserRoute()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error) # Récupération de
l'utilisateur à partir des informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error) # Repository
|   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool # Vérifie que la
session de l'utilisateur est valide
|   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le
refresh_token de l'utilisateur
|> func (s *Server) CreateUserRoute() http.HandlerFunc
# Handler HTTP
|> func (s *Service) CreateRouteForUser(ctx context.Context, user
*models.User, route *validations.RouteValidator) # Service
|   |> func (r *Routes) InsertRoute(ctx context.Context, route
*models.Route) # Repository -
création de la route
|> func RouteToDTO(route *models.Route) *RouteDTO
# Conversion DTO
```

```
↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error                                # Ecriture de la
réponse avec une fonction générique
```

PATCH /users/me/routes/{routeId}

Cet endpoint permet à un utilisateur authentifié de modifier un de ses itinéraires existants.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

Paramètre	Type	Description
routeId	int64	Identifiant de l'itinéraire

```
{
  "name": "string",
  "route": [
    {
      "lat": 0.0,
      "lon": 0.0
    },
    ...
  ]
}
```

Règles de validation :

- name : Requis, nouveau nom de l'itinéraire
- route : Requis, nouveau tableau de points contenant au moins une coordonnée
- route[].lat : Requis, latitude en degrés décimaux
- route[].lon : Requis, longitude en degrés décimaux

Réponse

```
{
  "id": 0,
  "name": "string",
  "route": [
    {
      "lat": 0.0,
      "lon": 0.0
    },
    ...
  ]
}
```

```

    ],
    "created_at": "string",
    "updated_at": "string"
}

```

Trace

```

mux.Handle("PATCH /users/me/routes/{routeId}", s.AuthMiddleware()
(s.PatchUserRoute()))
└─> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   └─> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error) #
Récupération de l'utilisateur à partir des informations de son token JWT
décodé
|   |   └─> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error) #
Repository
|   └─> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool # Vérifie
que la session de l'utilisateur est valide
|   └─> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le
refresh_token de l'utilisateur
└─> func (s *Server) PatchUserRoute() http.HandlerFunc
# Handler HTTP
|   └─> func (s *Service) PatchUserRoute(ctx context.Context, user
*models.User, routeId int64, route *validations.RouteValidator) # Service
|   └─> func (r *Routes) GetRouteUserById(ctx context.Context, userId,
routeId int64) # Repository -
vérifie l'existence de la route
|   └─> func (r *Routes) UpdateRoute(ctx context.Context, route
*models.Route) #
Repository - mise à jour de la route
|   └─> func RouteToDTO(route *models.Route) *RouteDTO
# Conversion DTO
|   └─> mathdeordr.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error #
Ecriture de la réponse avec une fonction générique

```

DELETE /users/me/routes/{routeId}

Cet endpoint permet à un utilisateur authentifié de supprimer un de ses itinéraires.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)

Paramètres / Corps de requête

Paramètre	Type	Description
routeId	int64	Identifiant de l'itinéraire

Aucun corps de requête n'est requis pour cette requête.

Réponse

Retourne un code 204 (No Content) en cas de succès.

Trace

```

mux.Handle("DELETE /users/me/routes/{routeId}", s.AuthMiddleware()
(s.DeleteUserRoute()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   |> func (s *Service) GetUserByID(ctx context.Context, id int64)
(*models.User, error)      # Récupération de l'utilisateur à partir des
informations de son token JWT décodé
|   |   |> func (u *Users) FindByID(ctx context.Context, id int64)
(*models.User, error)      # Repository
|   |   |> func (s *Service) IsAuthenticated(ctx context.Context, user
*models.User) bool        # Vérifie que la session de l'utilisateur est
valide
|   |   |> func (t *Tokens) Get(ctx context.Context, user *models.User)
(*models.Token, error) # Récupère le refresh_token de l'utilisateur
|   |   |> func (s *Server) DeleteUserRoute() http.HandlerFunc
# Handler HTTP
|   |   |> func (s *Service) DeleteRoute(ctx context.Context, routeId int64,
user *models.User)      # Service
|   |   |   |> func (r *Routes) GetRouteUserById(ctx context.Context, userId,
routeId int64)          # Repository - vérifie l'existence de la route
|   |   |   |> func (r *Routes) DeleteRoute(ctx context.Context, routeId,
userId int64)            # Repository - suppression de la route
|   |   |   |> mathdeordr.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error # Ecriture de la réponse avec une fonction
générique

```

GIS (Geographic Information System)

1. Introduction

Le microservice **supmap-gis** fournit des fonctionnalités avancées de traitement géographique pour l'écosystème Supmap : calcul d'itinéraires multimodaux, géocodage (et géocodage inverse), et prise en compte dynamique des incidents.

- Pour le routing, on utilise [Valhalla](#) qui est un projet open source qu'on auto-héberge.
- Pour le geocoding, on utilise [Nominatim](#), un projet open source qu'on auto-héberge également.

1.1. Rôles principaux

- **Exposer une API HTTP** : endpoints `/geocode`, `/address`, `/route`, `/health` via `net/http` (handlers personnalisés).
- **Routage multimodal** : calcul d'itinéraires avec options (type de véhicule, exclusions dynamiques, alternatives...).
- **Géocodage** : conversion d'adresses en coordonnées GPS (et inversement).
- **Prise en compte des incidents** : intégration du service interne `supmap-incidents` pour éviter des routes lors du calcul d'itinéraires.

1.2. Fonctionnement général

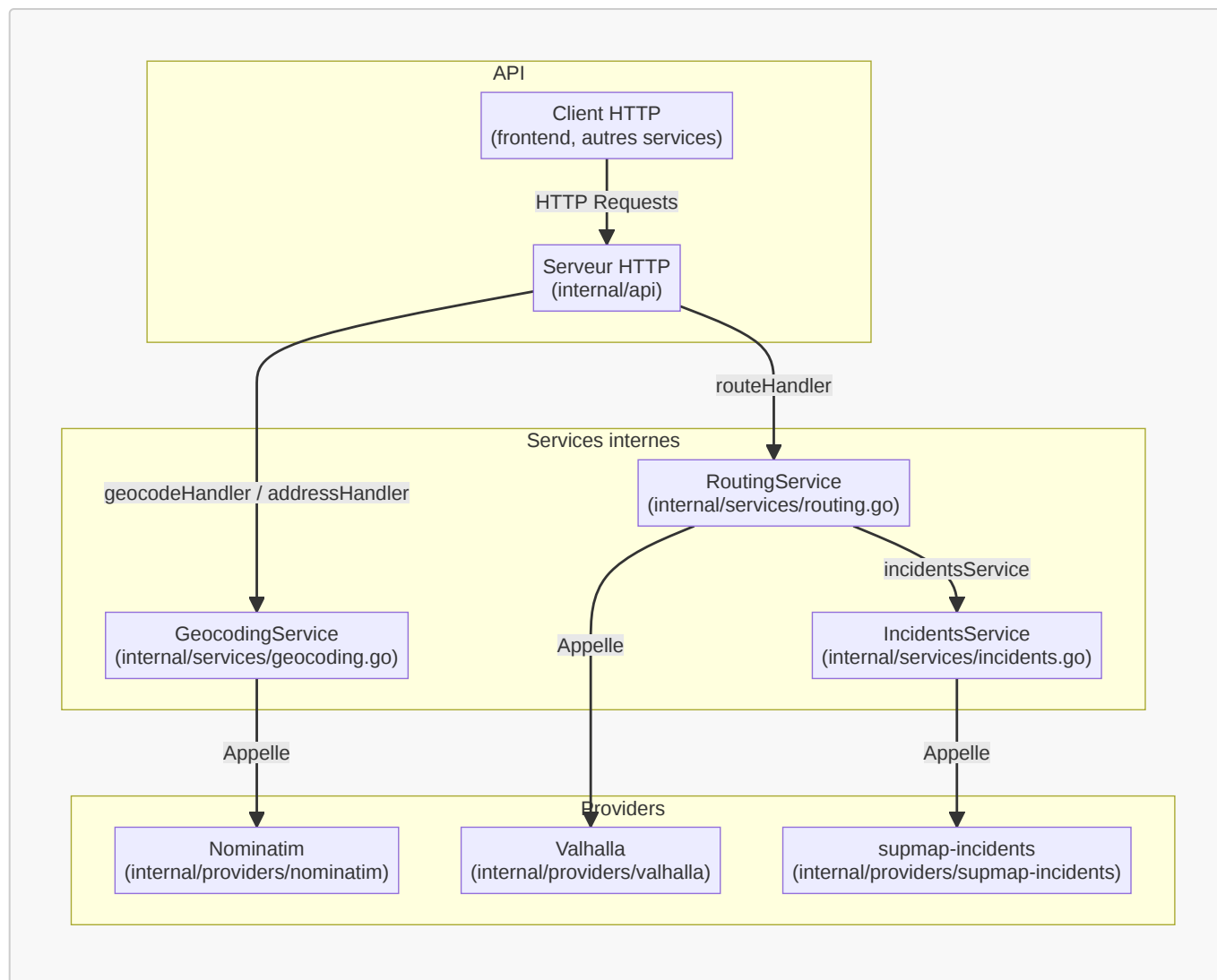
- **Entrée principale** : `cmd/api/main.go`
 - Instancie la configuration, le logger, les clients HTTP externes (Nominatim, Valhalla, `supmap-incidents`).
 - Compose les services métiers : `GeocodingService`, `IncidentsService`, `RoutingService`, injectés au serveur HTTP.
 - Lance l'API HTTP via `internal/api/server.go` (serveur, routing, doc Swagger).
- **Handlers HTTP** : `internal/api/handlers.go`
 - Chaque endpoint a un handler dédié, qui :
 - Valide les paramètres ou le corps de la requête.
 - Appelle le service métier correspondant.
 - Formate la réponse ou l'erreur.
- **Services métiers** : `internal/services/`
 - **GeocodingService** : encapsule les appels à Nominatim, standardise les résultats.
 - **RoutingService** : orchestre le calcul d'itinéraire avec Valhalla, l'enrichit en excluant dynamiquement les routes touchées par des incidents bloquants via `IncidentsService`.
 - **IncidentsService** : interroge `supmap-incidents` pour identifier les points à éviter.
 - **Polyline decoding utilitaire** : interprète les polylines Valhalla (tracé des trajets).
- **Providers** : `internal/providers/`
 - Clients HTTP spécialisés pour : Valhalla (routing), Nominatim (géocodage), `supmap-incidents` (incidents).

1.3. Technologies et principes d'architecture

- **Go natif** : `net/http`, pas de framework tiers lourd. On utilise la librairie standard.
 - **Injection de dépendances explicite** : chaque service reçoit ses clients et dépendances à l'instanciation.
 - **Découplage fort** : chaque handler et service métier a une responsabilité claire, testable et extensible.
 - **Contrôle des erreurs centralisé** : propagation explicite des erreurs vers le handler HTTP.
-

2. Architecture générale

2.1. Schéma d'architecture



2.2. Description des interactions internes et externes

- **Entrée dans le service** : Les requêtes HTTP arrivent sur le serveur (`internal/api/server.go`). Chaque endpoint (`/geocode`, `/address`, `/route`, `/health`) possède un handler dédié dans `internal/api/handlers.go`.
- **Géocodage** : Les handlers `/geocode` et `/address` font appel au `GeocodingService`. Celui-ci délègue les requêtes à un client Nominatim (provider interne) pour exécuter le géocodage direct ou inverse. Les résultats sont formatés et renvoyés au client.
- **Routage** : Le handler `/route` utilise le `RoutingService`. Avant de lancer le calcul d'itinéraire, ce service :
 1. Demande au `IncidentsService` si des incidents sont présents autour des points de passage (via le provider `supmap-incidents`).
 2. Exclut dynamiquement ces zones du calcul d'itinéraire.
 3. Lance la requête de routage auprès du provider `Valhalla`.
 4. Formate et retourne la réponse.
- **Gestion des incidents** : Le service `IncidentsService` encapsule la logique d'appel à l'API `supmap-incidents` : Il calcule un cercle englobant tous les points de départ/d'arrivée/intermédiaires et interroge le provider pour obtenir la liste des incidents à proximité.

- **Configuration et initialisation** : Le point d'entrée (`cmd/api/main.go`) :
 - Charge la configuration (`internal/config`)
 - Instancie les clients providers (Nominatim, Valhalla, supmap-incidents)
 - Compose les services métiers
 - Instancie le serveur HTTP

2.3. Présentation des principaux composants

- **Serveur HTTP** (`internal/api/server.go`)
 - Démarre le serveur, gère le routage des endpoints, fournit la documentation Swagger.
 - Injecte les services métiers nécessaires aux handlers.
- **Handlers HTTP** (`internal/api/handlers.go`)
 - Orchestrant la validation des entrées, l'appel aux services métiers et la gestion des erreurs/réponses.
- **Services métiers** (`internal/services/`)
 - `GeocodingService` : Encapsule la logique de géocodage via Nominatim.
 - `RoutingService` : Gère le calcul d'itinéraires, l'intégration des incidents, la transformation des réponses Valhalla.
 - `IncidentsService` : Calcule la zone à surveiller, interroge supmap-incidents, filtre les incidents pertinents qui nécessitent d'être évités.
- **Providers** (`internal/providers/`)
 - Contiennent les clients HTTP pour chaque service externe :
 - Nominatim (géocodage)
 - Valhalla (routage)
 - supmap-incidents (incidents)
- **Configuration** (`internal/config`)
 - Centralise la configuration du service (ports, hôtes, etc.).

3. Organisation du projet et Structure des dossiers

Le projet est organisé selon une architecture claire et modulaire inspirée des standards Go, favorisant la séparation des responsabilités et la testabilité.

3.1 Arborescence commentée (niveaux principaux)

```
supmap-gis/  
├─ cmd/  
│   └─ api/                # Point d'entrée du service (main.go)  
├─ internal/  
│   └─ api/                # Serveur HTTP, routing, handlers et
```

```
middlewares
|   └─ config/                # Chargement et validation de la configuration
(variables d'environnement)
|   └─ providers/            # Clients pour services externes (Valhalla,
Nominatim, supmap-incidents)
|   └─ services/            # Logique métier (géocodage, routage,
incidents, utilitaires)
└─ go.mod / go.sum          # Dépendances Go
└─ Dockerfile              # Déploiement containerisé
└─ ...                      # Autres fichiers (docs, configs, CI, etc.)
```

3.2 Rôle de chaque dossier/fichier principal

- **cmd/api/** Contient le point d'entrée du microservice (**main.go**).
- **internal/api/**
 - Définit le serveur HTTP, le routing (association endpoints/handlers), la gestion du CORS, la documentation Swagger.
 - Fichiers clés :
 - **server.go** : instantiation du serveur, mapping des routes
 - **handlers.go** : logique des endpoints (**/geocode**, **/address**, **/route**, **/health**)
 - **middleware.go** : middlewares, ex : gestion CORS
- **internal/config/**
 - Centralise le chargement et la validation de la configuration (hôtes, ports des providers, etc.)
 - Permet l'utilisation de variables d'environnement
- **internal/providers/**
 - Implémente un client HTTP pour chaque service tiers ou interne :
 - **nominatim/** : géocodage/adressage
 - **valhalla/** : routage
 - **supmap-incidents/** : incidents routiers
- **internal/services/**
 - Regroupe la logique métier :
 - **geocoding.go** : intégration et adaptation des résultats Nominatim
 - **routing.go** : orchestration du calcul d'itinéraire via Valhalla, gestion dynamique des exclusions (incidents)
 - **incidents.go** : interrogation et filtrage des incidents pertinents
 - **polyline.go** : utilitaires de décodage de polygones Valhalla
- **go.mod / go.sum** Gestion des dépendances et de la version Go du projet.

4. Détail des services internes

Cette section présente le fonctionnement interne des principaux services métier.

4.1. GeocodingService

- **Rôle** : Fournit les opérations de géocodage direct (adresse → coordonnées) et inverse (coordonnées → adresse "humaine"). Sert d'interface métier entre l'API et le provider Nominatim, en standardisant et en validant les résultats.
- **Dépendances** :
 - Client Nominatim (via l'interface `GeocodingClient`)
- **Principales méthodes**
 - `Search(ctx, address string) ([]Place, error)` → Appelle le provider, convertit et filtre les résultats Nominatim. → Retourne une liste de structures Place (lat, lon, nom, display_name).
 - `Reverse(ctx, lat, lon float64) (*nominatim.ReverseResult, error)` → Géocodage inverse, retourne la structure ReverseResult du provider.

4.2. RoutingService

- **Rôle** : Orchestration complète du calcul d'itinéraire :
 - Prise en compte des incidents à proximité ("zones à éviter" dynamiques)
 - Appel du provider Valhalla
 - Transformation et enrichissement de la réponse pour l'API (legs, summary, shape, alternatives...)
- **Dépendances** :
 - Client Valhalla (`RoutingClient`)
 - `IncidentsService` (pour lister les incidents autour du trajet)
- **Principales méthodes**
 - `CalculateRoute(ctx, routeRequest valhalla.RouteRequest) (*[]Trip, error)` → Extrait les points du trajet, interroge `IncidentsService`, enrichit la requête Valhalla en excluant les points à risque, appelle Valhalla, convertit la réponse (trips, legs, summary...)
 - Fonctions d'adaptation ("mapping") :
 - `MapValhallaTrip(vt valhalla.Trip) (*Trip, error)`
 - `mapValhallaLeg(vl valhalla.Leg) (*Leg, error)` → Transformations détaillées du format Valhalla vers les DTO internes.

4.3. IncidentsService

- **Rôle** : Fournit la liste des incidents routiers à prendre en compte lors du calcul d'itinéraire, selon les points de passage du trajet. Calcule un cercle englobant ("bounding circle") autour des points et interroge le provider supmap-incidents.
- **Dépendances** :

- Client supmap-incidents (**IncidentsClient**)

- **Principales méthodes**

- **IncidentsAroundLocations**(ctx, locations []Point) []Point → Calcule le centre et le rayon optimaux, appelle le provider, filtre les incidents pertinents nécessitant d'être évités.
- Fonctions utilitaires privées :
 - **computeLocationsBoundingCircle**(locations []Point) (centerLat, centerLon, radius)
 - **haversine**(lat1, lon1, lat2, lon2 float64) float64 (pour la distance sphérique)

4.4. Résumé des dépendances

- **GeocodingService** → Client Nominatim
- **RoutingService** → Client Valhalla, IncidentsService
- **IncidentsService** → Client supmap-incidents

L'instanciation des services se fait dans le **main.go**, chaque service recevant explicitement ses dépendances (découplage fort, testabilité).

Chaque service expose uniquement les méthodes nécessaires à ses usages métier, en cachant la complexité des providers et en garantissant un formatage homogène pour l'API.

5. Endpoints HTTP exposés

5.1. Tableau récapitulatif

Méthode	Chemin	Description fonctionnelle
GET	/geocode	Géocodage d'une adresse (adresse → coordonnées)
GET	/address	Géocodage inverse (coordonnées → adresse)
POST	/route	Calcul d'itinéraire multimodal avec exclusions
GET	/health	(Non documenté ici, endpoint de liveness/readiness)

5.2. Détails des endpoints

5.2.1. **/geocode** — Géocodage d'adresse

- **Méthode + chemin** **GET** **/geocode**
- **Description fonctionnelle** Convertit une adresse "humaine" en coordonnées GPS. Retourne une liste de résultats possibles (lat, lon, nom...).
- **Paramètres attendus**
 - Query : **address** (obligatoire) — l'adresse à géocoder

- Exemple de requête

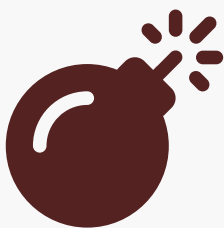
```
GET /geocode?address=Abbaye aux Dames Caen
```

- Exemple de réponse

```
{
  "data": [
    {
      "lat": 49.1864,
      "lon": -0.3608,
      "name": "Abbaye aux Dames",
      "display_name": "Abbaye aux Dames, Caen, France"
    },
    ...
  ],
  "message": "success"
}
```

- Description du flux de traitement

- Vérification du paramètre `address` (400 si manquant)
- Appel à `GeocodingService.Search()`
- Conversion et formatage des résultats
- Retour HTTP 200 avec la liste (vide si aucun résultat) ou 500 en cas d'erreur



Syntax error in text
mermaid version 10.4.0

5.2.2. /address — Géocodage inverse

- Méthode + chemin `GET /address`

- **Description fonctionnelle** Retourne l'adresse "humaine" la plus proche pour des coordonnées GPS fournies.

- Paramètres attendus

- Query : `lat` (obligatoire, float, ex: 49.0677)
- Query : `lon` (obligatoire, float, ex: -0.6658)

- Exemple de requête

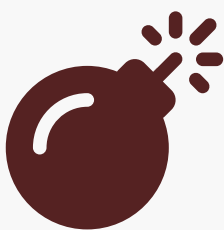
```
GET /address?lat=49.0677&lon=-0.6658
```

- **Exemple de réponse**

```
{
  "display_name": "Place de la Gare, Caen, France"
}
```

- **Description du flux de traitement**

- Vérification des paramètres `lat` et `lon` (400 si manquant)
- Appel à `GeocodingService.Reverse()`
- Extraction du champ `display_name` du premier résultat
- Retour 200 avec l'adresse, 404 si aucune trouvée



Syntax error in text
mermaid version 10.4.0

5.2.3. `/route` — Calcul d'itinéraire multimodal

- **Méthode + chemin** `POST /route`
- **Description fonctionnelle** Calcule l'itinéraire optimal selon un ensemble de points, le profil de déplacement, et les incidents à éviter (prise en compte dynamique des exclusions). Peut proposer des itinéraires alternatifs.
- **Paramètres attendus**
 - Body (JSON) :
 - `locations` (obligatoire, array) : liste d'objets `{lat, lon}` (au moins 2)
 - `costing` (obligatoire, string) : mode de transport (`auto`, `bicycle`, etc.)
 - `exclude_locations` (optionnel) : coordonnées à éviter (normalement gérées automatiquement)
 - `costing_options` (optionnel, objet) : options permettant d'éviter les péages, les ferries et les autoroutes
 - `language` (optionnel, string, défaut `fr-FR`) : langue des instructions
 - `alternates` (optionnel, int, défaut 2)
- **Exemple de requête**

```
POST /route
{
  "costing": "auto",
  "costing_options": {
    "use_tolls": 0
  },
  "locations": [
    {"lat": 49.1864, "lon": -0.3608},
    {"lat": 49.0677, "lon": -0.6658}
  ]
}
```

- Exemple de réponse

```
{
  "data": [
    {
      "locations": [...],
      "legs": [
        {
          "maneuvers": [
            {"instruction": "Prendre à droite", ...}
          ],
          "summary": {"length": 8.2, "time": 740},
          "shape": [
            {"lat": 49.18, "lon": -0.36},
            ...
          ]
        }
      ],
      "summary": {"length": 8.2, "time": 740}
    }
  ],
  "message": "success"
}
```

- Description du flux de traitement

- Décodage et validation du body JSON (locations >= 2, costing valide...)
- Conversion en requête Valhalla
- Appel à `RoutingService.CalculateRoute()`
 - Appel à `IncidentsService` pour exclure dynamiquement les incidents
 - Appel au provider Valhalla
 - Mapping du résultat (legs, maneuvers, summary...)
- Retour 200 avec la liste des itinéraires ou 500 en cas d'erreur



Syntax error in text

mermaid version 10.4.0

6. Structures & interfaces importantes

Cette section synthétise les structures et interfaces clés du projet, en complément des détails déjà vus sur les endpoints et services. Elle se concentre sur la modélisation métier, l'API, et le découplage via interfaces. (Les détails des champs sont volontairement réduits pour éviter la redite.)

6.1. Structures principales

6.1.1. Entités métier

- **Place**
 - Représente un résultat de géocodage (lat, lon, nom, display_name).
- **Trip / Leg / Maneuver / Summary**
 - Décomposent un itinéraire : un Trip regroupe une ou plusieurs Leg (tronçons), chacune contenant des Maneuver (instructions) et un résumé (Summary).
- **Point**
 - Simple couple latitude/longitude utilisé dans plusieurs contextes (requêtes, incidents, polylines...).

6.1.2. Structures de requête/réponse API

- **RouteRequest**
 - Body de `/route`. Contient : `locations`, `costing`, `exclude_locations`, `costing_options`, `language`, `alternates`.
- **AddressResponse**
 - Réponse de `/address`. Champ unique : `display_name`.
- **handler.Response[T]**
 - Enveloppe générique standardisant les réponses JSON (champ `data` et `message`).

6.2. Interfaces clés

6.2.1. Interfaces de clients (Providers)

- **GeocodingClient**
 - `Search(ctx, address string) ([]GeocodeResult, error)`
 - `Reverse(ctx, lat, lon float64) (*ReverseResult, error)`
- **RoutingClient**
 - `CalculateRoute(ctx, routeRequest) (*RouteResponse, error)`
- **IncidentsClient**

- `IncidentsInRadius(ctx, lat, lon, radius) ([]Incident, error)`

6.2.2. Interfaces de services métiers

- **GeocodingService**
 - `Search, Reverse`
- **RoutingService**
 - `CalculateRoute`
- **IncidentsService**
 - `IncidentsAroundLocations`

Chacune de ces interfaces permet d'injecter des implémentations alternatives (mocks, doubles, providers réels...) pour les tests ou l'évolution du projet.

6.3. Diagramme des dépendances principales



Syntax error in text
mermaid version 10.4.0

Les structures et interfaces sont conçues pour garantir un découplage fort, une testabilité maximale et une évolution facilitée du code métier et des intégrations externes.

7. Stack trace & appels typiques

Cette section détaille, pour chacun des endpoints principaux, la stack trace logique typique (ordre d'appel des fonctions/services), les signatures des fonctions clés, et fournit un diagramme de séquence synthétique pour la compréhension globale.

7.1. Stack trace typique : `/geocode` (GET)

Stack trace (ordre d'appel)

1. `Server.geocodeHandler()`
2. `GeocodingService.Search(ctx, address string) ([]Place, error)`
3. `GeocodingClient.Search(ctx, address string) ([]GeocodeResult, error)`

Signatures principales

- `func (s *Server) geocodeHandler() http.HandlerFunc`
- `func (s *GeocodingService) Search(ctx context.Context, address string) ([]Place, error)`
- `func (n *NominatimClient) Search(ctx context.Context, address string) ([]GeocodeResult, error)`

Diagramme de séquence



Syntax error in text
mermaid version 10.4.0

7.2. Stack trace typique : `/address` (GET)

Stack trace (ordre d'appel)

1. `Server.addressHandler()`
2. `GeocodingService.Reverse(ctx, lat, lon float64) (*ReverseResult, error)`
3. `GeocodingClient.Reverse(ctx, lat, lon float64) (*ReverseResult, error)`

Signatures principales

- `func (s *Server) addressHandler() http.HandlerFunc`
- `func (s *GeocodingService) Reverse(ctx context.Context, lat, lon float64) (*nominatim.ReverseResult, error)`
- `func (n *NominatimClient) Reverse(ctx context.Context, lat, lon float64) (*ReverseResult, error)`

Diagramme de séquence



Syntax error in text
mermaid version 10.4.0

7.3. Stack trace typique : `/route` (POST)

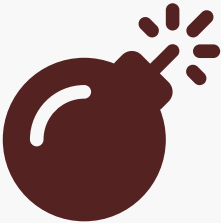
Stack trace (ordre d'appel)

1. `Server.routeHandler()`
2. `RoutingService.CalculateRoute(ctx, routeRequest valhalla.RouteRequest) ([]Trip, error)`
3. `IncidentsService.IncidentsAroundLocations(ctx, locations []Point) []Point`
4. `IncidentsClient.IncidentsInRadius(ctx, lat, lon, radius) ([]Incident, error)`
5. `RoutingClient.CalculateRoute(ctx, routeRequest) (*RouteResponse, error)`
6. Mapping functions (`MapValhallaTrip`, etc.)

Signatures principales

- `func (s *Server) routeHandler() http.HandlerFunc`
- `func (s *RoutingService) CalculateRoute(ctx context.Context, routeRequest valhalla.RouteRequest) ([]Trip, error)`
- `func (s *IncidentsService) IncidentsAroundLocations(ctx context.Context, locations []Point) []Point`
- `func (c *IncidentsClient) IncidentsInRadius(ctx context.Context, lat, lon float64, radius RadiusMeter) ([]Incident, error)`
- `func (c *ValhallaClient) CalculateRoute(ctx context.Context, req valhalla.RouteRequest) (*valhalla.RouteResponse, error)`

Diagramme de séquence simplifié



Syntax error in text

mermaid version 10.4.0

8. Configuration et build

8.1. Variables d’environnement

Le service utilise des variables d’environnement pour la configuration ; elles sont typiquement définies dans un fichier `.env` (à la racine du projet ou injectées dans l’environnement d’exécution).

Variable	Rôle
<code>API_SERVER_HOST</code>	Hôte d’écoute du serveur API HTTP
<code>API_SERVER_PORT</code>	Port d’écoute du serveur API HTTP
<code>NOMINATIM_HOST</code>	Hôte du provider Nominatim (géocodage)
<code>NOMINATIM_PORT</code>	Port du provider Nominatim
<code>VALHALLA_HOST</code>	Hôte du provider Valhalla (routage)
<code>VALHALLA_PORT</code>	Port du provider Valhalla
<code>SUPMAP_INCIDENTS_HOST</code>	Hôte du provider supmap-incidents
<code>SUPMAP_INCIDENTS_PORT</code>	Port du provider supmap-incidents

Exemple de fichier `.env` :

```
API_SERVER_HOST=0.0.0.0
API_SERVER_PORT=8080
NOMINATIM_HOST=nominatim
```

```
NOMINATIM_PORT=8081
VALHALLA_HOST=valhalla
VALHALLA_PORT=8002
SUPMAP_INCIDENTS_HOST=supmap-incidents
SUPMAP_INCIDENTS_PORT=8082
```

8.2. CI : build & push automatique (GitHub Actions)

Le dépôt embarque un workflow CI/CD (.github/workflows/image-publish.yml) qui :

- **Déclencheur** : sur chaque push sur la branche **master**
- **Actions** :
 1. Checkout du code
 2. Login au registre de conteneurs GitHub (**ghcr.io**) via **GITHUB_TOKEN**
 3. Build de l'image Docker (taggée **ghcr.io/4proj-le-projet-d-une-vie/supmap-gis:latest**)
 4. Push de l'image sur GitHub Container Registry

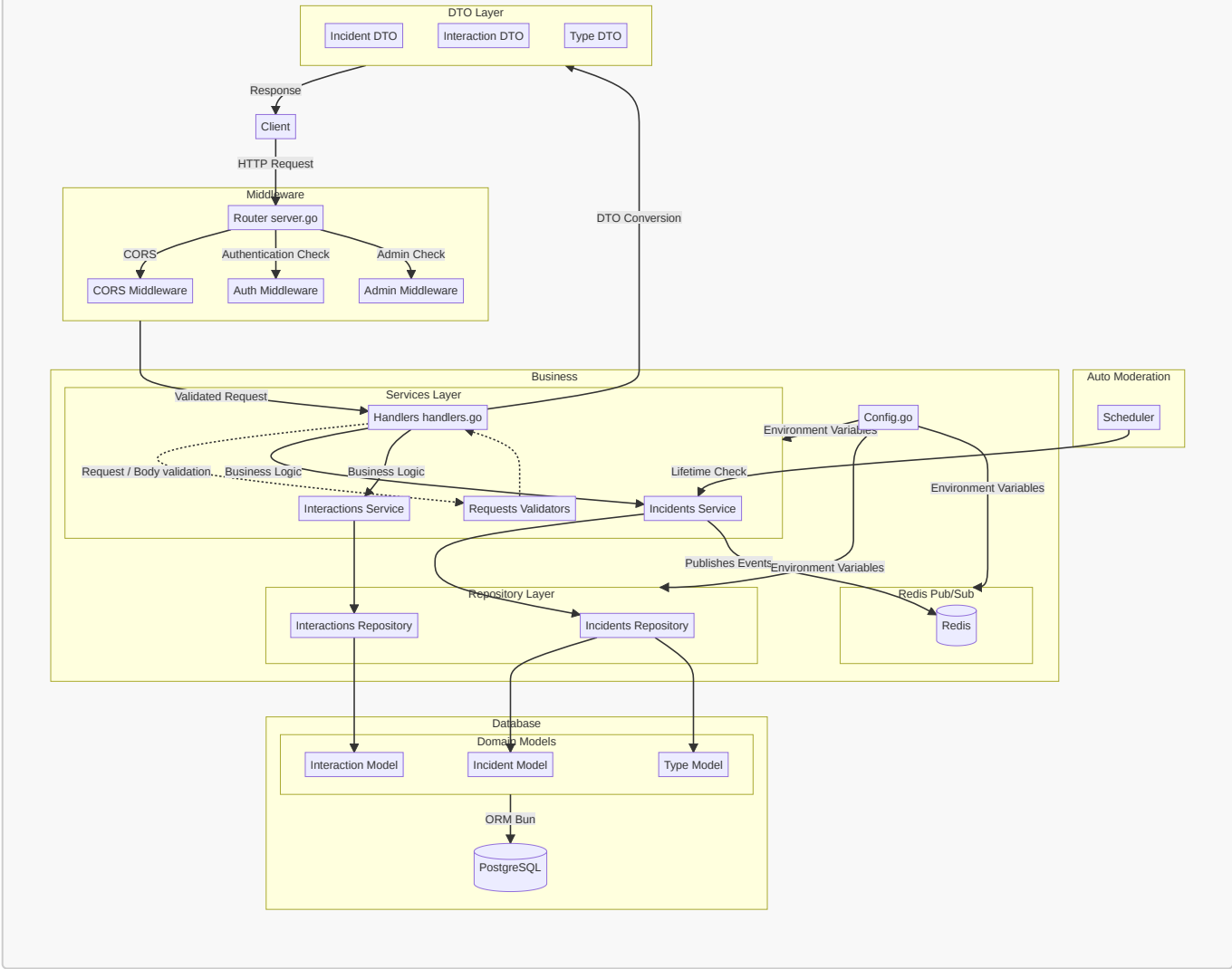
Incidents

Microservice de gestion des incidents pour Supmap

Présentation

supmap-incidents est un microservice écrit en Go destiné à la gestion des incidents de navigation pour Supmap.

Architecture



```
supmap-incidents/  
├── cmd/  
│   └── api/  
│       └── main.go  
microservice  
├── internal/  
│   ├── api/  
│   │   └── handlers.go  
HTTP  
│   └── server.go  
et routes  
│   ├── middleware.go  
│   └── validations/  
│       └── ...  
│   └── config/  
│       └── config.go  
d'environnement  
│   ├── models/  
│   └── dto/  
données  
│   └── ...  
l'ORM Bun  
│   └── repository/  
requêtes SQL avec l'ORM Bun
```

Point d'entrée du
Gestionnaires de requêtes
Configuration du serveur HTTP
Intercepteurs de requête
Structures de validation
Configuration et variables
DTOs permettant d'exposer les
Structures de données pour
Repository implémentant les

```

|   |   |   | ...
|   |   |   | services/
fonctionnalités métier du service # Services implémentant les
|   |   |   |   | ...
|   |   |   |   | redis/
|   |   |   |   |   | redis.go
|   |   |   |   |   | messages.go
pub/sub # Configuration du client Redis
|   |   |   |   |   | # Messages envoyés dans le
|   |   |   |   |   | scheduler/
|   |   |   |   |   |   | scheduler.go
à intervalle régulier # Service appelant une fonction
|   |   |   |   |   |   | auto-moderate-incidents.go # Fonctions d'auto modération
|   |   |   |   |   |   | # Documentation Swagger auto
|   |   |   |   |   | docs/
implémentée avec Swggo # Image Docker du microservice
|   |   |   |   |   |   | Dockerfile
|   |   |   |   |   |   | go.mod
|   |   |   |   |   |   | go.sum /
(auto généré) # Dépendances Go
|   |   |   |   |   |   | # Checksums des dépendances
|   |   |   |   |   |   | README.md
|   |   |   |   |   |   | # Documentation du projet

```

Prérequis et installation

- Go 1.24
- Base de données postgres (conteneurisée ou non)

Démarrage rapide

```

# Cloner le repo
git clone https://github.com/4PROJ-Le-projet-d-une-vie/supmap-incidents.git
cd supmap-incidents

# Démarrer le service (nécessite les variables d'environnement, voir ci-dessous)
go run ./cmd/api

```

Avec Docker

```

docker pull ghcr.io/4proj-le-projet-d-une-vie/supmap-incidents:latest
docker run --env-file .env -p 8080:80 supmap-incidents

```

Authentification

Pour pull l'image, il faut être authentifié par docker login.

- Générer un Personal Access Token sur GitHub :
 - Se rendre sur <https://github.com/settings/tokens>

- Cliquer sur "Generate new token"
- Cocher au minimum la permission read:packages
- Copier le token
- Connecter Docker à GHCR avec le token :

```
echo 'YOUR_GITHUB_TOKEN' | docker login ghcr.io -u YOUR_GITHUB_USERNAME --password-stdin
```

Configuration

La configuration se fait via des variables d'environnement ou un fichier `.env` :

Variable	Description
ENV	Définit l'environnement dans lequel est exécuté le programme (par défaut production)
DB_URL	URL complète vers la base de donnée
PORT	Port sur lequel écoutera le service pour recevoir les requêtes
SUPMAP_USERS_HOST	Host du service utilisateur
SUPMAP_USERS_PORT	Port du service utilisateur sur la machine host
REDIS_HOST	Host du service redis
REDIS_PORT	Port du service redis sur la machine host
REDIS_INCIDENTS_CHANNEL	Nom du channel du pub/sub redis dans lequel sont publiés les messages (par défaut incidents)

Swagger

Chaque handler de ce service comprend des commentaires [Swaggo](#) pour créer dynamiquement une page Swagger-ui. Exécutez les commande suivantes pour générer la documentation :

```
# Installez l'interpréteur de commande Swag
go install github.com/swaggo/swag/cmd/swag@latest

# Générez la documentation
swag init -g cmd/api/main.go
```

Maintenant, vous pouvez accéder à l'URL <http://localhost:8080/swagger/index.html> décrivant la structure attendue pour chaque endpoint de l'application

NB: La documentation n'inclut pas les endpoints `/internal` destinés à une utilisation exclusivement interne

Authentification

L'authentification dans ce service est entièrement externalisée vers le microservice [supmap-users](#).

Lorsqu'une requête authentifiée arrive sur le service, le middleware d'authentification :

1. Récupère le token JWT depuis le header [Authorization](#)
2. Effectue une requête HTTP vers le endpoint interne [/internal/users/check-auth](#) du service utilisateurs
3. Vérifie la réponse :
 - Si le token est valide, la requête continue son traitement
 - Si le token est invalide ou expiré, une erreur 401 ou 403 est retournée

Cette approche permet de :

- Centraliser la logique d'authentification dans un seul service
- Garantir la cohérence des vérifications de sécurité
- Simplifier la maintenance en évitant la duplication de code

Note: Les routes [/internal](#) ne sont accessibles que depuis le réseau interne et ne nécessitent pas d'authentification supplémentaire

Migrations de base de données

Les migrations permettent de versionner la structure de la base de données et de suivre son évolution au fil du temps. Elles garantissent que tous les environnements (développement, production, etc.) partagent le même schéma de base de données.

Ce projet utilise [Goose](#) pour gérer les migrations SQL. Les fichiers de migration sont stockés dans le dossier [migrations/changelog/](#) et sont embarqués dans le binaire grâce à la directive [//go:embed](#) dans [migrate.go](#).

Création d'une migration

Pour créer une nouvelle migration, installez d'abord le CLI Goose :

```
go install github.com/pressly/goose/v3/cmd/goose@latest
```

Puis créez une nouvelle migration (la commande se base sur les variables du fichier `.env`) :

```
# Crée une migration vide
goose -dir migrations/changelog create nom_de_la_migration sql

# La commande génère un fichier horodaté dans migrations/changelog/
# Exemple: 20240315143211_nom_de_la_migration.sql
```

Exécution des migrations

Les migrations sont exécutées automatiquement au démarrage du service via le package migrations :

```
// Dans main.go
if err := migrations.Migrate("pgx", conf.DbUrl, logger); err != nil {
    logger.Error("migration failed", "err", err)
}
```

Le package migrations utilise embed.FS pour embarquer les fichiers SQL dans le binaire :

```
//go:embed changelog/*.sql
var changelog embed.FS
// Connexion à la base de données
goose.SetBaseFS(changelog)
```

Auto-modération des incidents

Le service intègre un système d'auto-modération asynchrone qui vérifie et nettoie périodiquement les incidents selon des règles métier définies.

Fonctionnement du scheduler

Le scheduler utilise un `time.Ticker` pour exécuter des tâches de modération à intervalles réguliers de manière asynchrone :

```
func (s *Scheduler) Run() {
    go func() {
        for {
            select {
            case <-s.ticker.C: // Channel bloquant qui reçoit un signal à
chaque tick
                s.CheckLifetimeWithoutConfirmation(ctx, tx)
                s.CheckGlobalLifeTime(ctx, tx)
            case <-s.stop:      // Channel permettant d'arrêter proprement
le scheduler
                s.ticker.Stop()
                return
            }
        }
    }()
}
```

Le scheduler complet est dans [scheduler.go](#)

Le scheduler est lancé dans une goroutine (`go func()`) pour s'exécuter de manière asynchrone sans bloquer le reste de l'application. Le select attend alors deux types d'événements :

- Un signal du ticker à intervalle régulier pour lancer les vérifications
- Un signal d'arrêt pour terminer proprement le scheduler

Règles de modération

Deux types de vérifications sont effectuées sur les incidents actifs :

Durée sans confirmation :

```
noInteractionThreshold :=  
time.Duration(incident.Type.LifetimeWithoutConfirmation)  
if time.Since(incident.UpdatedAt) > noInteractionThreshold*time.Second {  
    // Suppression de l'incident  
}
```

Si un incident n'a reçu aucune interaction pendant une durée définie par son type, il est automatiquement supprimé.

Durée de vie globale :

```
incidentTTL := time.Duration(incident.Type.GlobalLifetime)  
if time.Since(incident.CreatedAt) > incidentTTL*time.Second {  
    // Suppression de l'incident  
}
```

Chaque type d'incident définit une durée de vie maximale. Une fois cette durée dépassée, l'incident est automatiquement supprimé.

Lorsqu'un incident est supprimé par l'auto-modération, un message est publié dans Redis pour notifier les autres services :

```
err = s.redis.PublishMessage(s.config.IncidentChannel,  
&rediss.IncidentMessage{  
    Data:    *dto.IncidentToRedis(&incident),  
    Action: rediss.Deleted,  
})
```

Communication par Redis Pub/Sub

Redis est utilisé dans ce service comme un système de messagerie en temps réel grâce à son mécanisme de Publish/Subscribe (Pub/Sub). Cette approche permet de notifier les autres services du système lors de changements d'état des incidents.

Fonctionnement du Pub/Sub Redis

Le Pub/Sub est un pattern de messagerie où les émetteurs (publishers) envoient des messages dans des canaux spécifiques, sans connaissance directe des destinataires. Les récepteurs (subscribers) s'abonnent aux canaux qui les intéressent pour recevoir ces messages.

Dans notre application, nous utilisons ce mécanisme pour publier trois types d'événements :

```
const (  
    Create      Action = "create"    // Nouvel incident créé  
    Certified Action = "certified" // Incident certifié par suffisamment  
d'interactions positives  
    Deleted     Action = "deleted"   // Incident supprimé (manuellement ou  
par auto-modération)  
)
```

Implémentation

Le service utilise un client Redis asynchrone qui publie les messages via un channel Go :

```
type Redis struct {  
    client *redis.Client  
    send   chan redis.Message // Channel pour les messages à envoyer  
}  
  
func (r *Redis) Run(ctx context.Context) {  
    go r.publisher(ctx) // Démarre le publisher dans une goroutine  
}  
  
func (r *Redis) publisher(ctx context.Context) {  
    for {  
        select {  
            case msg := <-r.send: // Attend les messages à publier  
                err := r.client.Publish(ctx, msg.Channel, msg.Payload).Err()  
                if err != nil {  
                    r.log.Error("redis publish message error", "error", err)  
                }  
            case <-ctx.Done(): // Arrêt propre lors de la fermeture du  
service  
                return  
        }  
    }  
}
```

Structure des messages

Les messages publiés suivent une structure commune :

```
type IncidentMessage struct {  
    Data    dto.IncidentRedis `json:"data"`    // Données de l'incident  
    Action Action            `json:"action"`  // Type d'action  
    (create/certified/deleted)  
}
```

Cette approche permet aux autres services de réagir en temps réel aux changements d'état des incidents, permettant la mise à jour des interfaces utilisateur en cours de navigation.

Gestion des transactions SQL concurrentes

Dans un environnement distribué où plusieurs instances du service peuvent être déployées, la gestion de la concurrence est cruciale pour maintenir l'intégrité des données.

Problématique

Le service étant stateless et scalable horizontalement, plusieurs scénarios problématiques peuvent survenir :

- Plusieurs utilisateurs interagissent simultanément avec le même incident
- L'auto-modération s'exécute pendant qu'un utilisateur interagit avec un incident
- Les données lues peuvent devenir obsolètes entre le moment de la lecture et de l'écriture

Sans gestion de la concurrence, ces situations peuvent mener à :

- Des données corrompues ou incohérentes
- Des règles d'auto-modération appliquées sur des données périmées
- Des mises à jour perdues ou écrasées

Solution : Transactions avec verrouillage

Le service utilise des transactions SQL avec la clause **FOR UPDATE** qui permet de verrouiller les enregistrements pendant leur modification. Voici un exemple simplifié :

```
// Récupération d'une transaction  
tx, err := s.incidents.AskForTx(ctx)  
if err != nil {  
    return nil, err  
}  
// Relachement de la transaction à la fin de la fonction  
defer func() {  
    if err != nil {  
        s.log.Info("Rollback de la transaction")  
        _ = tx.Rollback()  
    } else {  
        s.log.Info("Commit de la transaction")  
        _ = tx.Commit()  
    }  
}()  
}
```

```
// Verrouille l'enregistrement pour les autres transactions
exec.NewSelect().
    Model(&incident).
    For("UPDATE").    // Équivalent SQL: SELECT ... FOR UPDATE
    Scan(ctx)
```

Fonctionnement

Quand une transaction démarre :

1. Elle pose un verrou sur l'incident concerné
 - Les autres requêtes souhaitant modifier ou consulter (avec FOR UPDATE) cet incident sont mises en attente
 - La transaction effectue ses modifications (interactions, auto-modération)
 - Une fois la transaction terminée, le verrou est libéré
 - Les requêtes en attente sont alors traitées séquentiellement :
2. Chacune accède aux données dans leur état le plus récent
 - Les règles métier s'appliquent sur des données cohérentes
 - L'intégrité des données est garantie même avec plusieurs instances du service

Endpoints

Les endpoints ci-dessous sont présentés selon l'ordre dans lequel ils sont définis dans [server.go](#)

GET /incidents

Get endpoint permet de trouver tous les incidents dans un rayon autour d'un point. Il est possible de filtrer par type d'incident.

Authentification / Autorisations

Aucune authentification n'est nécessaire, cet endpoint est public.

Paramètres / Corp de requête

Paramètre	Type	Description
lat	float64	Latitude du point central de la zone de recherche
lon	float64	Longitude du point central de la zone de recherche
radius	int64	Rayon en mètres dans lequel seront cherchés les incidents
include	string	Valeurs possibles : <ul style="list-style-type: none">- interactions (inclut toutes les interactions de l'incident)- summary (inclut un résumé des interactions de l'incident)

Réponse

include non défini

```
[
  {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    },
    "type": {
      "id": 0,
      "name": "string",
      "description": "string",
      "need_recalculation": true
    },
    "lat": 0,
    "lon": 0,
    "created_at": "string",
    "deleted_at": "string",
    "updated_at": "string",
    "distance": 0
  },
  ...
]
```

include définit à [interactions](#)

```
[
  {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    },
    "type": {
      "description": "string",
      "id": 0,
      "name": "string",
      "need_recalculation": true
    },
    "lat": 0,
    "lon": 0,
    "interactions": [
```

```

    {
      "id": 0,
      "user": {
        "handle": "string",
        "id": 0,
        "role": {
          "name": "string"
        }
      },
      "is_still_present": true,
      "created_at": "string"
    },
    ...
  ],
  "created_at": "string",
  "updated_at": "string",
  "deleted_at": "string",
  "distance": 0
},
...
]

```

Interactions définit à [summary](#)

```

[
  {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    },
    "type": {
      "description": "string",
      "id": 0,
      "need_recalculation": true,
      "name": "string"
    },
    "lat": 0,
    "lon": 0,
    "interactions_summary": {
      "is_still_present": 0,
      "no_still_present": 0,
      "total": 0
    },
    "created_at": "string",
    "updated_at": "string",
    "deleted_at": "string",
    "distance": 0
  }
]

```



```
    },
    ...
]
```

Trace

```
mux.Handle("GET /incidents", s.GetAllInRadius())
↳ func (s *Server) GetAllInRadius() http.HandlerFunc
# Handler HTTP
    ↳ func (s *Service) FindIncidentsInRadius(ctx context.Context, typeId
*int64, lat, lon float64, radius int64) ([]models.IncidentWithDistance,
error) # Service
    | ↳ func (i *Incidents) FindIncidentTypeById(ctx context.Context,
id *int64) (*models.Type, error)
# Repository
    | ↳ func (i *Incidents) FindIncidentsInZone(ctx context.Context,
lat, lon *float64, radius int64, typeId *int64)
([]models.IncidentWithDistance, error) # Repository
    | ↳ func (i *Incidents) FindIncidentById(ctx context.Context, id
int64) (*models.Incident, error)
# Repository
    | ↳ func (i *Incidents) FindIncidentByIdTx(ctx context.Context,
exec bun.IDB, id int64) (*models.Incident, error)
# Repository (Inclut une gestion de transactions concurrentes)
    ↳ func IncidentWithDistanceToDTO(incident
*models.IncidentWithDistance, interactionsState InteractionsResultState)
*IncidentWithDistanceDTO # Conversion DTO
    ↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error
# Ecriture de la réponse avec une fonction générique
```

GET /incidents/me/history

Récupère l'historique des incidents créés par l'utilisateur authentifié qui ont été supprimés (auto-modération ou interactions négatives).

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- Une session valide est requise (sinon code http 403)

Paramètres / Corps de requête

Paramètre	Type	Description
Valeurs possibles :		
include	string	- interactions (inclut toutes les interactions de l'incident) - summary (inclut un résumé des interactions de l'incident)

Réponse

include non défini

```
[
  {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    },
    "type": {
      "id": 0,
      "name": "string",
      "description": "string",
      "need_recalculation": true
    },
    "lat": 0,
    "lon": 0,
    "created_at": "string",
    "deleted_at": "string",
    "updated_at": "string",
    "distance": 0
  },
  ...
]
```

include définit à [interactions](#)

```
[
  {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    },
    "type": {
      "description": "string",
      "id": 0,
      "name": "string",
      "need_recalculation": true
    },
    "lat": 0,
```

```

    "lon": 0,
    "interactions": [
      {
        "id": 0,
        "user": {
          "handle": "string",
          "id": 0,
          "role": {
            "name": "string"
          }
        },
        "is_still_present": true,
        "created_at": "string"
      },
      ...
    ],
    "created_at": "string",
    "updated_at": "string",
    "deleted_at": "string",
    "distance": 0
  },
  ...
]

```

Interactions définit à [summary](#)

```

[
  {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    },
    "type": {
      "description": "string",
      "id": 0,
      "need_recalculation": true,
      "name": "string"
    },
    "lat": 0,
    "lon": 0,
    "interactions_summary": {
      "is_still_present": 0,
      "no_still_present": 0,
      "total": 0
    },
    "created_at": "string",
    "updated_at": "string",
  }
]

```

```

    "deleted_at": "string",
    "distance": 0
  },
  ...
]
```

Trace

```

mux.Handle("GET /incidents/me/history", s.AuthMiddleware()
(s.GetUserHistory()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   ↳ GET /internal/users/check-auth
# Vérification du token par le service users
↳ func (s *Server) GetUserHistory() http.HandlerFunc
# Handler HTTP
|> func (s *Service) GetUserHistory(ctx context.Context, user
*dto.PartialUserDTO) ([]models.Incident, error)      # Service
|   ↳ func (i *Incidents) FindUserHistory(ctx context.Context, user
*dto.PartialUserDTO) ([]models.Incident, error)      # Repository
|> func IncidentToDTO(incident *models.Incident, interactionsState
InteractionsResultState) *IncidentDTO                # Conversion DTO
↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error                          # Ecriture de la
réponse
```

GET /incidents/types

Récupère la liste de tous les types d'incidents disponibles dans l'application.

Authentification / Autorisations

Aucune authentification requise, endpoint public.

Paramètres / Corps de requête

Aucun paramètre requis.

Réponse

```

[
  {
    "id": 0,
    "name": "string",
    "description": "string",
    "need_recalculation": true
  },
]
```

```
    ...  
]
```

Trace

```
mux.Handle("GET /incidents/types", s.GetIncidentsTypes())  
↳ func (s *Server) GetIncidentsTypes() http.HandlerFunc  
# Handler HTTP  
    ↳ func (s *Service) GetAllIncidentTypes(ctx context.Context)  
    ([]models.Type, error) # Service  
    |    ↳ func (i *Incidents) FindAllIncidentTypes(ctx context.Context)  
    ([]models.Type, error) # Repository  
    ↳ func TypeToDTO(type *models.Type) *TypeDTO  
# Conversion DTO  
    ↳ mathdeodrd.handler/func Encode[T any](v T, status int, w  
http.ResponseWriter) error # Écriture de la réponse
```

GET /incidents/types/{id}

Récupère les détails d'un type d'incident spécifique par son ID.

Authentification / Autorisations

Aucune authentification requise, endpoint public.

Paramètres / Corps de requête

Paramètre	Type	Description
id	int64	ID du type d'incident recherché

Réponse

```
{  
  "id": 0,  
  "name": "string",  
  "description": "string",  
  "need_recalculation": true  
}
```

Trace

```
mux.Handle("GET /incidents/types/{id}", s.GetIncidentTypeById())  
↳ func (s *Server) GetIncidentTypeById() http.HandlerFunc
```

```
# Handler HTTP
|> func (s *Service) FindTypeById(ctx context.Context, id int64)
(*models.Type, error) # Service
|   |> func (i *Incidents) FindIncidentTypeById(ctx context.Context,
id *int64) (*models.Type, error) # Repository
|> func TypeToDTO(type *models.Type) *TypeDTO
# Conversion DTO
|> mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error # Ecriture de la réponse
```

POST /incidents

Crée un nouvel incident ou ajoute une interaction positive à un incident existant si un incident similaire existe déjà dans un rayon de 100m.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- Une session valide est requise (sinon code http 403)
- Un utilisateur ne peut pas créer plus d'un incident par minute (code http 429)

Paramètres / Corps de requête

```
{
  "type_id": 0,
  "lat": 0,
  "lon": 0
}
```

Règles de validation :

- type_id : ID d'un type d'incident existant
- lat : Latitude entre -90 et 90
- lon : Longitude entre -180 et 180

Réponse

```
{
  "id": 0,
  "user": {
    "handle": "string",
    "id": 0,
    "role": {
      "name": "string"
    }
  },
  "type": {
```

```

    "description": "string",
    "id": 0,
    "name": "string",
    "need_recalculation": true
  },
  "lat": 0,
  "lon": 0,
  "created_at": "string",
  "updated_at": "string",
  "deleted_at": "string"
}

```

Trace

```

mux.Handle("POST /incidents", s.AuthMiddleware()(s.CreateIncident()))
|> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   ↳ GET /internal/users/check-auth
# Vérification du token par le service users
↳ func (s *Server) CreateIncident() http.HandlerFunc
# Handler HTTP
|> func (s *Service) CreateIncident(ctx context.Context, user
*dto.PartialUserDTO, body *validations.CreateIncidentValidator)
(*models.Incident, error)      # Service
|   |> func (i *Incidents) FindIncidentTypeById(ctx context.Context,
id *int64) (*models.Type, error)
# Repository
|   |> func (i *Incidents) GetLastUserIncident(ctx context.Context,
user *dto.PartialUserDTO) (*models.Incident, error)
# Repository
|   |> func (i *Incidents) FindIncidentsInZone(ctx context.Context,
lat, lon *float64, radius int64, typeId *int64)
([]models.IncidentWithDistance, error)  # Repository
|   |> func (i *Incidents) FindIncidentById(ctx context.Context, id
int64) (*models.Incident, error)
# Repository
|   |> func (i *Incidents) CreateIncident(ctx context.Context,
incident *models.Incident) error
# Repository (Inclut une gestion de transactions concurrentes)
|   ↳ func (r *Redis) PublishMessage(channel string, payload any)
error
# Publication de l'événement Redis
|> func IncidentToDTO(incident *models.Incident, interactionsState
InteractionsResultState) *IncidentDTO
# Conversion DTO
↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error
# Ecriture de la réponse

```

Permet à un utilisateur d'interagir avec un incident existant en confirmant ou infirmant sa présence.

Authentification / Autorisations

- L'utilisateur doit être authentifié (sinon code http 401)
- Une session valide est requise (sinon code http 403)
- Un utilisateur ne peut pas interagir avec son propre incident (code http 403)
- Un utilisateur ne peut pas interagir plus d'une fois par heure avec le même incident (code http 429)

Paramètres / Corps de requête

```
{
  "incident_id": 0,
  "is_still_present": true
}
```

Règles de validation :

- incident_id : ID d'un incident existant
- is_still_present : Booléen indiquant si l'incident est toujours présent

Réponse

include non défini

```
{
  "id": 0,
  "user": {
    "handle": "string",
    "id": 0,
    "role": {
      "name": "string"
    }
  },
  "incident": {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    }
  },
  "type": {
    "description": "string",
    "id": 0,
    "name": "string",
    "need_recalculation": true
  }
}
```



```
    },
    "lat": 0,
    "lon": 0,
    "created_at": "string",
    "updated_at": "string",
    "deleted_at": "string"
  },
  "created_at": "string",
  "is_still_present": true
}
```

include définit à [interactions](#)

```
{
  "id": 0,
  "user": {
    "handle": "string",
    "id": 0,
    "role": {
      "name": "string"
    }
  },
  "incident": {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    }
  },
  "type": {
    "description": "string",
    "id": 0,
    "name": "string",
    "need_recalculation": true
  },
  "lat": 0,
  "lon": 0,
  "interactions": [
    "string"
  ],
  "created_at": "string",
  "updated_at": "string",
  "deleted_at": "string"
},
"created_at": "string",
"is_still_present": true
}
```

Interactions définit à [summary](#)

```
{
  "id": 0,
  "user": {
    "handle": "string",
    "id": 0,
    "role": {
      "name": "string"
    }
  },
  "incident": {
    "id": 0,
    "user": {
      "handle": "string",
      "id": 0,
      "role": {
        "name": "string"
      }
    }
  },
  "type": {
    "description": "string",
    "id": 0,
    "name": "string",
    "need_recalculation": true
  },
  "lat": 0,
  "lon": 0,
  "interactions_summary": {
    "is_still_present": 0,
    "no_still_present": 0,
    "total": 0
  },
  "created_at": "string",
  "updated_at": "string",
  "deleted_at": "string"
},
"created_at": "string",
"is_still_present": true
}
```

Trace

```
mux.Handle("POST /incidents/interactions", s.AuthMiddleware()
(s.UserInteractWithIncident()))
└─> func (s *Server) AuthMiddleware() func(http.Handler) http.Handler
# Authentifie l'utilisateur
|   └─> GET /internal/users/check-auth
# Vérification du token par le service users
```

```

↳ func (s *Server) UserInteractWithIncident() http.HandlerFunc
# Handler HTTP
    ↳ func (s *Service) CreateInteraction(ctx context.Context, user
*dto.PartialUserDTO, body *validations.CreateInteractionValidator)
(*models.Interaction, error) # Service
    |   ↳ func (i *Incidents) FindIncidentByIdTx(ctx context.Context, tx
bun.IDB, id int64) (*models.Incident, error)
# Repository avec transaction
    |   ↳ func (i *Interactions) InsertTx(ctx context.Context, exec
bun.IDB, interaction *models.Interaction) error
# Repository avec transaction
    |   ↳ func (i *Incidents) UpdateIncidentTx(ctx context.Context, exec
bun.IDB, incident *models.Incident) error
# Repository avec transaction
    |   ↳ func (i *Interactions) FindInteractionByIdTx(ctx
context.Context, exec bun.IDB, id int64) (*models.Interaction, error)
# Repository avec transaction
    |   ↳ func (r *Redis) PublishMessage(channel string, payload any)
error
# Publication de l'événement Redis
    ↳ func InteractionToDTO(interaction models.Interaction,
interactionsState InteractionsResultState) *InteractionDTO
# Conversion DTO
    ↳ mathdeodrd.handler/func Encode[T any](v T, status int, w
http.ResponseWriter) error
# Ecriture de la réponse

```

Navigation

1. Introduction

1.1. Rôle du microservice

supmap-navigation est le microservice dédié à la gestion de la navigation en temps réel pour les utilisateurs de l'application Supmap. Il établit et maintient des connexions WebSocket avec les clients mobiles afin de :

- Suivre en direct la position de chaque utilisateur pendant leur trajet.
- Diffuser instantanément les nouveaux incidents signalés sur leur itinéraire.
- Gérer les recalculs de route à la volée en cas d'événement perturbateur (ex : accident, embouteillage).

1.2. Principales responsabilités

- **Connexion WebSocket et gestion de session** : Chaque client ouvre une connexion WebSocket identifiée par un `session_id` unique (UUID). Le serveur conserve en cache les informations de navigation et les positions des clients grâce à Redis.
- **Suivi de position** : Les clients envoient régulièrement leur position. Le service met à jour le cache et peut ainsi déterminer à tout moment l'avancement de l'utilisateur sur son trajet.
- **Diffusion d'incidents en temps réel** : Lorsqu'un nouvel incident est détecté ou modifié (via le microservice supmap-incidents), supmap-navigation est notifié via un canal Pub/Sub Redis. Il transmet

alors en temps réel l'incident aux clients concernés, c'est-à-dire ceux dont l'itinéraire croise la zone de l'incident.

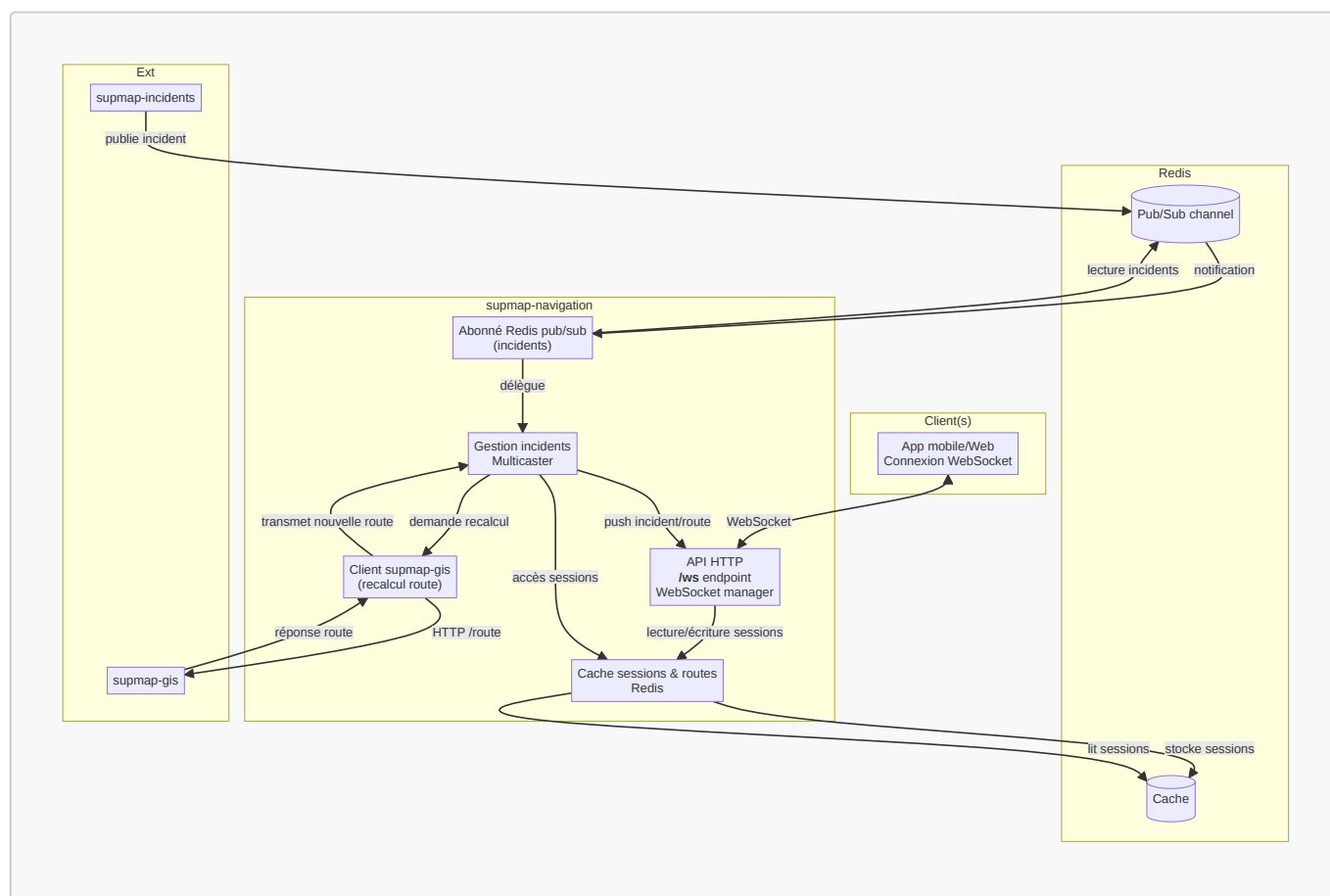
- **Recalcul dynamique des itinéraires** : Si un incident nécessite le recalcul de la route (incident bloquant et certifié...), le service interroge supmap-gis pour obtenir un nouvel itinéraire. Ce nouvel itinéraire est ensuite envoyé au(x) client(s) via la connexion WebSocket, assurant une navigation optimisée en permanence.

1.3. Technologies et dépendances externes

- **Go** : langage principal du microservice.
- **WebSocket** : communication temps réel bidirectionnelle avec les clients.
- **Redis** :
 - Stockage temporaire (cache) des sessions, routes et positions clients.
 - Mécanisme Pub/Sub pour recevoir en direct les incidents depuis supmap-incidents.
- **supmap-gis** : microservice utilisé pour le recalcul d'itinéraires en cas d'incident bloquant.
- **supmap-incidents** : source des incidents signalés sur le réseau via Redis Pub/Sub.
- **GitHub Actions** : CI pour build/push l'image Docker sur le registre GHCR du repo.

2. Architecture générale

2.1. Schéma d'architecture



2.2. Description des interactions internes et externes

- **Client (mobile)** : Ouvre une connexion WebSocket sur `/ws` avec un `session_id` (UUID généré). Envoie ses infos d'itinéraire et régulièrement sa position.
- **API HTTP / WebSocket manager** : Gère l'ouverture, le cycle de vie et la fermeture des connexions WebSocket. Chaque client correspond à une session identifiée et mappée sur une instance interne.
- **Cache Redis** :
 - Stocke les sessions de navigation : dernière position, itinéraire courant.
 - Permet de récupérer l'état d'une session à tout instant, pour tous les modules (manager, incidents...).
- **Abonné Redis Pub/Sub** :
 - S'abonne au canal d'incidents publié par **supmap-incidents**.
 - À la réception d'un message d'incident, délègue la gestion au **Multicaster**.
- **Gestionnaire d'incidents (Multicaster)** :
 - Détermine quels clients (sessions actives) sont concernés par l'incident.
 - Push l'incident en temps réel uniquement aux clients concernés via WebSocket.
 - Si l'incident nécessite un recalcul de route, il interroge le service **supmap-gis**, met à jour la session et push la nouvelle route au(x) client(s) impacté(s).
- **Client supmap-gis** :
 - Interagit avec le microservice **supmap-gis** via HTTP pour recalculer un itinéraire si besoin (en cas d'incident bloquant et certifié).
- **supmap-incidents** :
 - Publie les incidents sur le canal Pub/Sub Redis, ce qui déclenche la chaîne de notifications côté navigation.

2.3. Présentation des principaux composants

- **API HTTP/WebSocket (internal/api, internal/ws)** : Expose l'unique endpoint `/ws` pour la navigation temps réel ; chaque nouvelle connexion est gérée comme un client identifié (`session_id`).
- **Cache Redis (internal/cache)** : Abstraction pour stocker et lire les objets de session. TTL configurable.
- **Gestionnaire d'incidents (internal/incidents/multicaster.go)** : Logique pour déterminer si un incident touche un client, envoyer la notification et déclencher le recalcul de route si nécessaire.
- **Abonné Pub/Sub (internal/subscriber)** : S'abonne au canal Redis des incidents, déserialise les messages et transmet au multicaster.
- **Client GIS (internal/gis/routing/client.go)** : Client HTTP vers supmap-gis pour demander un recalcul d'itinéraire.

- **Session navigation (internal/navigation)** : Struct représentant l'état d'une navigation en cours : route, position, timestamps...

3. Organisation du projet et Structure des dossiers

3.1. Arborescence commentée

```

supmap-navigation/
├── cmd/
│   └── main.go                # Point d'entrée du microservice
├── internal/
│   ├── api/                  # API HTTP : serveur, handler, routing
│   │   └── handler.go        # Handler du endpoint /ws (connexion
│   │                           WebSocket)
│   │   └── server.go          # Démarrage et gestion du serveur HTTP
│   │   └── cache/             # Cache des sessions de navigation (Redis)
│   │   └── redis.go           # Abstraction pour stocker/récupérer les
│   │                           sessions navigation
│   └── config/               # Chargement, parsing de la configuration
│       └── gis/               # Fonctions géospatiales & client supmap-
│                               gis
│   └── polyline.go           # Calculs/distances sur polylines (utile
│       incidents)
│   └── routing/
│       └── client.go          # Client HTTP pour interroger supmap-gis
│                               (recalcul d'itinéraire)
│   └── incidents/
│       └── multicaster.go     # Gestion de la diffusion des incidents
│                               # Multicast incidents/nouvelles routes aux
│                               clients concernés
│   └── navigation/
│       ├── routes, points... # Structures de navigation (sessions,
│       └── session.go         # Structs : Session, Position, Route,
│                               Point, etc.
│   └── subscriber/
│       └── subscriber.go      # Abonné Redis Pub/Sub aux incidents
│                               # Logique d'abonnement et de dispatch au
│                               multicaster
│   └── types.go               # Types pour désérialiser les messages
│   └── incidents
│       └── ws/                # Gestion WebSocket : clients, manager,
│                               messaging
│   └── client.go              # Logique d'un client WebSocket
│       (lifecycle, messaging)
│   └── manager.go             # Manager central des clients WebSocket
└── ...

```

3.2. Rôle de chaque dossier/fichier principal

3.2.1. cmd/

- **main.go** Point d'entrée du service : instancie la config, connecte Redis, démarre les managers, serveurs et workers.

3.2.2. internal/api/

- **server.go** Serveur HTTP principal, expose `/ws` (WebSocket) et `/health`.
- **handler.go** Handler pour la connexion WebSocket, gestion du handshake et vérification du paramètre `session_id`.

3.2.3. internal/cache/

- **redis.go** Abstraction pour stocker/récupérer une session navigation dans Redis (opérations Set/Get/Delete).

3.2.4. internal/config/

- Chargement et parsing des variables d'environnement (hôtes, ports, Redis, etc).

3.2.5. internal/gis/

- **polyline.go** Fonctions utilitaires pour les calculs géospatiaux (distance point-polyline, etc).
- **routing/client.go** Client HTTP pour appeler supmap-gis lors du recalcul d'itinéraire.

3.2.6. internal/incidents/

- **multicaster.go** Logique de multicasting des incidents :
 - Vérifie si un incident concerne la route d'un client.
 - Push l'incident à la session concernée.
 - Déclenche un recalcul de route si besoin.

3.2.7. internal/navigation/

- **session.go** Structures métier pour une session de navigation (Session, Position, Route, Point, etc).

3.2.8. internal/subscriber/

- **subscriber.go** S'abonne au canal Redis Pub/Sub des incidents, déserialise les messages, relaie au multicaster.
- **types.go** Types pour la désérialisation des messages incidents reçus.

3.2.9. internal/ws/

- **manager.go** Manager WebSocket central :
 - Gère l'ensemble des clients connectés.
 - Dispatch les messages (broadcast, ciblé...).
 - Enregistrement/déconnexion.
- **client.go** Représentation d'un client WebSocket individuel :
 - Gestion du lifecycle, envoi/réception de messages, ping/pong.

4. Détail des services internes

4.1. Gestion des sessions de navigation (`internal/navigation`, `internal/cache`)

4.1.1. Rôle

- Représente l'état de navigation d'un utilisateur : itinéraire courant, dernière position, timestamp de mise à jour.
- Permet de persister et de retrouver à tout instant l'état d'une session (utile pour la diffusion des incidents, le recalcul de route, etc).

4.1.2. Dépendances

- **Redis** (via `internal/cache/redis.go`) pour le stockage temporaire des sessions.
- Utilisé par le WebSocket manager, le multicaster d'incidents et le subscriber.

4.1.3. Principales méthodes

- **SessionCache (interface) :**
 - `SetSession(ctx, session) error` : Ajoute ou met à jour une session en cache.
 - `GetSession(ctx, sessionID) (*Session, error)` : Récupère l'état d'une session via son ID.
 - `DeleteSession(ctx, sessionID) error` : Supprime la session du cache.

4.2. WebSocket Manager et clients (`internal/ws`)

4.2.1. Rôle

- Gère toutes les connexions WebSocket actives (un client = une session).
- Assure l'inscription/désinscription des clients, le broadcast des messages, et la gestion fine des canaux (ping/pong, déconnexions...).
- Route les messages reçus côté client (init, position) et côté serveur (incident, route recalculée).

4.2.2. Dépendances

- S'appuie sur le cache session pour la persistance et la cohérence des états utilisateurs.
- Interagit avec le multicaster d'incidents pour pousser les messages incidents/routes.

4.2.3. Principales méthodes

- **Manager :**
 - `Start()` : Boucle principale, écoute inscriptions/désinscriptions/messages.
 - `Broadcast(message)` : Broadcast d'un message à tous les clients.
 - `HandleNewConnection(id, conn)` : Création et démarrage d'un nouveau client WebSocket.
 - `ClientsUnsafe(), RLock(), RUnlock()` : Gestion thread-safe des clients.
- **Client :**
 - `Start()` : Démarre les goroutines de lecture/écriture pour la connexion.

- `Send(msg)` : Envoie un message (avec gestion du buffer, déconnexion si bloqué).
- `handleMessage(msg)` : Routage des messages reçus (init, position...).

4.3. API HTTP (`internal/api`)

4.3.1. Rôle

- Expose l'endpoint `/ws` (WebSocket) et `/health` (vérification de vie).
- Effectue la première validation (`session_id`), puis délègue la gestion de la connexion au WebSocket manager.

4.3.2. Dépendances

- WebSocket manager (gestion des connexions)

4.3.3. Principales méthodes

- **Server** :
 - `Start(ctx)` : Démarrage du serveur HTTP, gestion propre du shutdown.
 - `wsHandler()` : Handler HTTP pour l'upgrade WebSocket (contrôle du paramètre `session_id`).

4.4. Cache Redis (`internal/cache`)

4.4.1. Rôle

- Fournit un cache persistant et performant pour les sessions de navigation.
- Permet de stocker, récupérer et supprimer l'état d'une session utilisateur.

4.4.2. Dépendances

- Client Redis (github.com/redis/go-redis/v9)
- Utilisé par le WebSocket manager, le multicaster d'incidents, et le subscriber.

4.4.3. Principales méthodes/fonctions

- `NewRedisSessionCache(client, ttl)` : Constructeur de la structure cache.
- `SetSession(ctx, session)` / `GetSession(ctx, sessionID)` / `DeleteSession(ctx, sessionID)` : Opérations CRUD sur les sessions.

4.5. Abonné Pub/Sub Redis (`internal/subscriber`)

4.5.1. Rôle

- S'abonne au canal Pub/Sub Redis où sont publiés les incidents par le microservice supmap-incidents.
- Déséréalise les messages incidents et délègue au multicaster la notification aux clients concernés.

4.5.2. Dépendances

- Client Redis
- Multicaster d'incidents

4.5.3. Principales méthodes/fonctions

- `Start(ctx)` : Boucle d'abonnement au canal Redis, gestion du pool de workers pour traiter les incidents.
- `handleMessage(ctx, msg)` : Désérialisation et dispatch d'un message incident au multicaster.

4.6. Gestionnaire/MultiDiffuseur d'Incidents (`internal/incidents/multicaster.go`)

4.6.1. Rôle

- Détermine dynamiquement quels clients sont concernés par un incident (en fonction de la route).
- Envoie l'incident (ou le recalcul d'itinéraire) en temps réel uniquement aux clients concernés.
- Si besoin, déclenche un recalcul d'itinéraire via le client GIS et met à jour la session.

4.6.2. Dépendances

- WebSocket manager (pour accéder à tous les clients connectés)
- SessionCache (pour lire/mettre à jour les routes)
- Client GIS (pour le recalcul d'itinéraire)

4.6.3. Principales méthodes/fonctions

- `MulticastIncident(ctx, incident, action)` : Parcourt tous les clients, détecte qui est concerné et leur push le bon message.
- `isIncidentOnRoute(incident, session)` : Vérifie la proximité de l'incident sur la route du client.
- `handleRouteRecalculation(ctx, client, session)` : Gère l'appel GIS, update la session, push la nouvelle route.
- `sendIncident(client, incident, action)` : Push un message incident à un client.

4.7. Client GIS (`internal/gis/routing`)

4.7.1. Rôle

- Communique avec le microservice supmap-gis pour recalculer des itinéraires.
- Utilisé lors de la certification d'un incident bloquant.

4.7.2. Dépendances

- HTTP Client standard
- supmap-gis (microservice)

4.7.3. Principales méthodes/fonctions

- `NewClient(baseURL)` : Instancie le client GIS.

- `CalculateRoute(ctx, routeRequest)` : Fait un POST `/route` à supmap-gis, récupère et déséréalise la réponse.

5. Endpoint HTTP exposé

5.1. Tableau récapitulatif

Méthode	Chemin	Description	Paramètres obligatoires
GET	/ws	Connexion WebSocket pour navigation temps réel	<code>session_id</code> (query)

5.2. Détail de l'endpoint `/ws`

5.2.1. Description fonctionnelle

L'endpoint `/ws` permet à un client (mobile) d'ouvrir une connexion WebSocket persistante avec le service **supmap-navigation** afin de :

- Suivre et mettre à jour sa position en temps réel.
- Recevoir des notifications d'incidents sur son itinéraire.
- Être notifié immédiatement d'un recalcul d'itinéraire si nécessaire.

Chaque connexion WebSocket correspond à une session de navigation unique, identifiée par un identifiant `session_id` fourni par le client (UUID généré côté client).

5.2.2. Méthode + chemin

- **Méthode** : `GET`
- **Chemin** : `/ws`

5.2.3. Paramètres d'ouverture

Type	Nom	Emplacement	Obligatoire	Description
query	<code>session_id</code>	Query	Oui	Identifiant unique de la session (UUID côté client)

Aucun header particulier n'est requis.

5.2.4. Exemple d'ouverture de connexion

Requête WebSocket (HTTP Upgrade) :

```
GET /ws?session_id=123e4567-e89b-12d3-a456-426614174000 HTTP/1.1
Host: navigation.supmap.local
Connection: Upgrade
Upgrade: websocket
Origin: https://app.supmap.local
Sec-WebSocket-Key: xxxxx==
```

```
Sec-WebSocket-Version: 13
```

```
...
```

Code JS côté client (exemple, ce n'est pas le vrai code) :

```
const sessionId = '123e4567-e89b-12d3-a456-426614174000';
const ws = new WebSocket(`wss://navigation.supmap.local/ws?
session_id=${sessionId}`);

ws.onopen = () => {
  // Envoi du message "init" contenant la route et la position
};
```

5.3. Description du flux de traitement

Résumé textuel :

1. **Connexion** : Le client tente d'ouvrir une connexion WebSocket sur `/ws` en passant son `session_id` en query.
2. **Validation** : Le serveur vérifie la présence du paramètre `session_id`. Si absent, la connexion est refusée.
3. **Upgrade et gestion** : Si OK, la connexion est acceptée, un client WebSocket est instancié et enregistré auprès du manager.
4. **Échange initial** : Le client envoie un message `init` contenant sa route et sa position actuelle.
5. **Traitement en continu** :
 - Le client envoie périodiquement des positions (`position`).
 - Le serveur push incidents et nouveaux itinéraires si besoin.
 - La connexion reste ouverte tant que la session est active (ou jusqu'à déconnexion).
6. **Déconnexion** : À la fermeture, le client est désinscrit du manager, la connexion WebSocket est fermée proprement.

5.4. Diagramme de séquence du traitement



Syntax error in text
mermaid version 10.4.0

6. Protocole & messages WebSocket

6.1. Tableau récapitulatif des types de messages

Sens	Type	Description
Client → Serveur	init	Initialisation de la session (route, position)
Client → Serveur	position	Envoi périodique de la position
Serveur → Client	incident	Notification d'un incident impactant l'itinéraire
Serveur → Client	route	Transmission d'un nouvel itinéraire recalculé

6.2. Structure générale des messages

Tout message échangé a la forme :

```
{
  "type": "TYPE_MESSAGE",
  "data": { ... }
}
```

- **type** : Type du message (voir tableau ci-dessus)
- **data** : Données associées, dont la structure dépend du type

6.3. Messages Client → Serveur

6.3.1. **init**

But : Initialiser la session côté serveur (première connexion), transmettre la route et la position courante du client.

Structure attendue :

```
{
  "type": "init",
  "data": {
    "session_id": "e38d5757-5359-44b3-ab6e-8c619e3daba0",
    "last_position": {
      "lat": 49.171669,
      "lon": -0.582579,
      "timestamp": "2025-05-06T20:52:30Z"
    },
    "route": {
      "polyline": [
        { "latitude": 49.171669, "longitude": -0.582579 },
        ...,
        { "latitude": 49.201345, "longitude": -0.392996 }
      ],
      "locations": [
        { "lat": 49.17167279051877, "lon": -0.5825858234777268 },
        { "lat": 49.20135359834111, "lon": -0.3930605474075204 }
      ]
    }
  }
}
```

```
    },  
    "updated_at": "2025-05-06T20:52:30Z"  
  }  
}
```

Description des champs :

- **session_id** : UUID de la session (identique à celui passé lors de la connexion WebSocket)
- **last_position** : Position actuelle (lat, lon, timestamp)
- **route.polyline** : Liste des points composant la polyline de l'itinéraire
- **route.locations** : Points d'arrêt (départ, arrivée, étapes)
- **updated_at** : Date de la dernière mise à jour de la session

6.3.2. position

But : Mise à jour de la position courante du client à intervalles réguliers (ex : toutes les cinq secondes).

Structure attendue :

```
{  
  "type": "position",  
  "data": {  
    "lat": 49.1943057668118,  
    "lon": -0.44595408906894096,  
    "timestamp": "2025-05-07T10:07:00Z"  
  }  
}
```

Description des champs :

- **lat, lon** : Coordonnées GPS de la position courante
- **timestamp** : Date et heure de la mesure

6.4. Messages Serveur → Client

6.4.1. incident

But : Notifier le client qu'un incident a été signalé, supprimé ou certifié sur son itinéraire.

Structure attendue :

```
{  
  "type": "incident",  
  "data": {  
    "incident": {  
      "id": 26,  
      "user_id": 2,  
      "type": {
```

```

    "id": 3,
    "name": "Embouteillage",
    "description": "Circulation fortement ralentie ou à l'arrêt.",
    "need_recalculation": true
  },
  "lat": 49.19477822,
  "lon": -0.3964915,
  "created_at": "2025-05-09T14:57:36.96141Z",
  "updated_at": "2025-05-09T14:57:36.96141Z"
},
"action": "create"
}
}

```

Description des champs :

- **incident**: Objet décrivant l'incident
 - **id**: Identifiant unique
 - **user_id**: Utilisateur ayant signalé l'incident
 - **type**: Type d'incident (nom, description, recalcul requis...)
 - **lat, lon**: Position de l'incident
 - **created_at, updated_at**: Dates de création/mise à jour
 - **deleted_at** (optionnel) : Date de suppression si l'incident est supprimé
- **action**: "create", "certified" ou "deleted"

6.4.2. route

But : Transmettre un nouvel itinéraire recalculé à la suite d'un incident bloquant certifié.

Structure attendue :

```

{
  "type": "route",
  "data": {
    "route": {
      "locations": [
        { "lat": 49.194305, "lon": -0.445954, "type": "break",
"original_index": 0 },
        { "lat": 49.201353, "lon": -0.39306, "type": "break",
"original_index": 1 }
      ],
      "legs": [
        {
          "maneuvers": [
            {
              "type": 1,
              "instruction": "Conduisez vers l'est sur N 13/E 46.",
              "street_names": [ "N 13", "E 46" ],
              "time": 11.75,
              "length": 0.293,

```

```

        "begin_shape_index": 0,
        "end_shape_index": 1
    },
    ...,
    {
        "type": 4,
        "instruction": "Vous êtes arrivé à votre destination.",
        "street_names": [],
        "time": 0, "length": 0,
        "begin_shape_index": 297,
        "end_shape_index": 297
    }
],
"summary": { "time": 448.775, "length": 6.349 },
"shape": [
    { "latitude": 49.194309, "longitude": -0.445953 },
    ...,
    { "latitude": 49.201345, "longitude": -0.392996 }
]
},
"summary": { "time": 448.775, "length": 6.349 }
},
"info": "recalculated_due_to_incident"
}
}

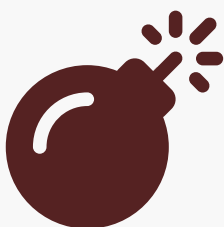
```

Description des champs :

- **route**: Nouvel itinéraire complet (même structure que lors du calcul initial avec supmap-gis)
- **info**: Raison du recalcul (ex : "recalculated_due_to_incident")

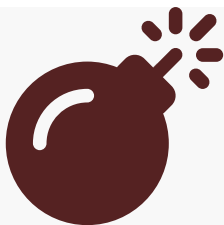
6.5. Flux typiques et diagrammes de séquence

6.5.1. Flux initialisation et suivi de position



Syntax error in text
mermaid version 10.4.0

6.5.2. Flux notification d'incident et recalcul d'itinéraire



Syntax error in text

mermaid version 10.4.0

7. Structures & interfaces importantes

7.1. Structures principales

7.1.1. Session de navigation ([internal/navigation/session.go](#))

```
type Session struct {
    ID            string    `json:"session_id"`
    LastPosition  Position  `json:"last_position"`
    Route         Route     `json:"route"`
    UpdatedAt     time.Time `json:"updated_at"`
}
type Position struct {
    Lat      float64 `json:"lat"`
    Lon      float64 `json:"lon"`
    Timestamp time.Time `json:"timestamp"`
}
type Route struct {
    Polyline []Point    `json:"polyline"`
    Locations []Location `json:"locations"`
}
type Point struct {
    Lat float64 `json:"latitude"`
    Lon float64 `json:"longitude"`
}
type Location struct {
    Lat float64 `json:"lat"`
    Lon float64 `json:"lon"`
}
```

- **Usage** : représente l'état complet d'une navigation utilisateur (position, itinéraire, timestamps, etc).

7.1.2. Message WebSocket ([internal/ws/client.go](#))

```
type Message struct {
    Type string    `json:"type"`
    Data json.RawMessage `json:"data"`
}
```

- **Usage** : enveloppe tout message WebSocket échangé (type + payload générique).

7.1.3. Client WebSocket ([internal/ws/client.go](#))

```
type Client struct {
    ID      string
    Conn    *websocket.Conn
    Manager *Manager
    send    chan Message
    ctx     context.Context
    cancel  context.CancelFunc
}
```

- **Usage** : représente une connexion WebSocket active côté serveur (1 client = 1 session de navigation).

7.1.4. Incident et payload ([internal/incidents/types.go](#))

```
type IncidentPayload struct {
    Incident *Incident `json:"incident"`
    Action  string    `json:"action"`
}
type Incident struct {
    ID          int64      `json:"id"`
    UserID      int64      `json:"user_id"`
    Type        *Type      `json:"type"`
    Lat         float64    `json:"lat"`
    Lon         float64    `json:"lon"`
    CreatedAt   time.Time  `json:"created_at"`
    UpdatedAt   time.Time  `json:"updated_at"`
    DeletedAt   *time.Time `json:"deleted_at,omitempty"`
}
type Type struct {
    ID          int64      `json:"id"`
    Name        string      `json:"name"`
    Description  string      `json:"description"`
    NeedRecalculation bool      `json:"need_recalculation"`
}
```

- **Usage** : incidents de circulation, utilisés pour notifier le client et déclencher un éventuel recalcul de route.

7.1.5. Manager WebSocket ([internal/ws/manager.go](#))

```
type Manager struct {
    clients    map[string]*Client
    register   chan *Client
    unregister chan *Client
}
```

```

    broadcast      chan Message
    mu             sync.RWMutex
    ctx            context.Context
    cancel         context.CancelFunc
    logger         *slog.Logger
    sessionCache   navigation.SessionCache
}

```

- **Usage** : composant central qui pilote tous les clients WebSocket et la diffusion des messages.

7.2. Interfaces métier utiles

7.2.1. SessionCache ([internal/navigation/session.go](#))

```

type SessionCache interface {
    SetSession(ctx context.Context, session *Session) error
    GetSession(ctx context.Context, sessionID string) (*Session, error)
    DeleteSession(ctx context.Context, sessionID string) error
}

```

- **Usage** : abstraction pour le cache des sessions (implémentée par Redis, mais testable/mockable).

7.2.2. Multicaster d'incidents ([internal/incidents/multicaster.go](#))

```

type Multicaster struct {
    Manager      *ws.Manager
    SessionCache navigation.SessionCache
    RoutingClient *routing.Client
}

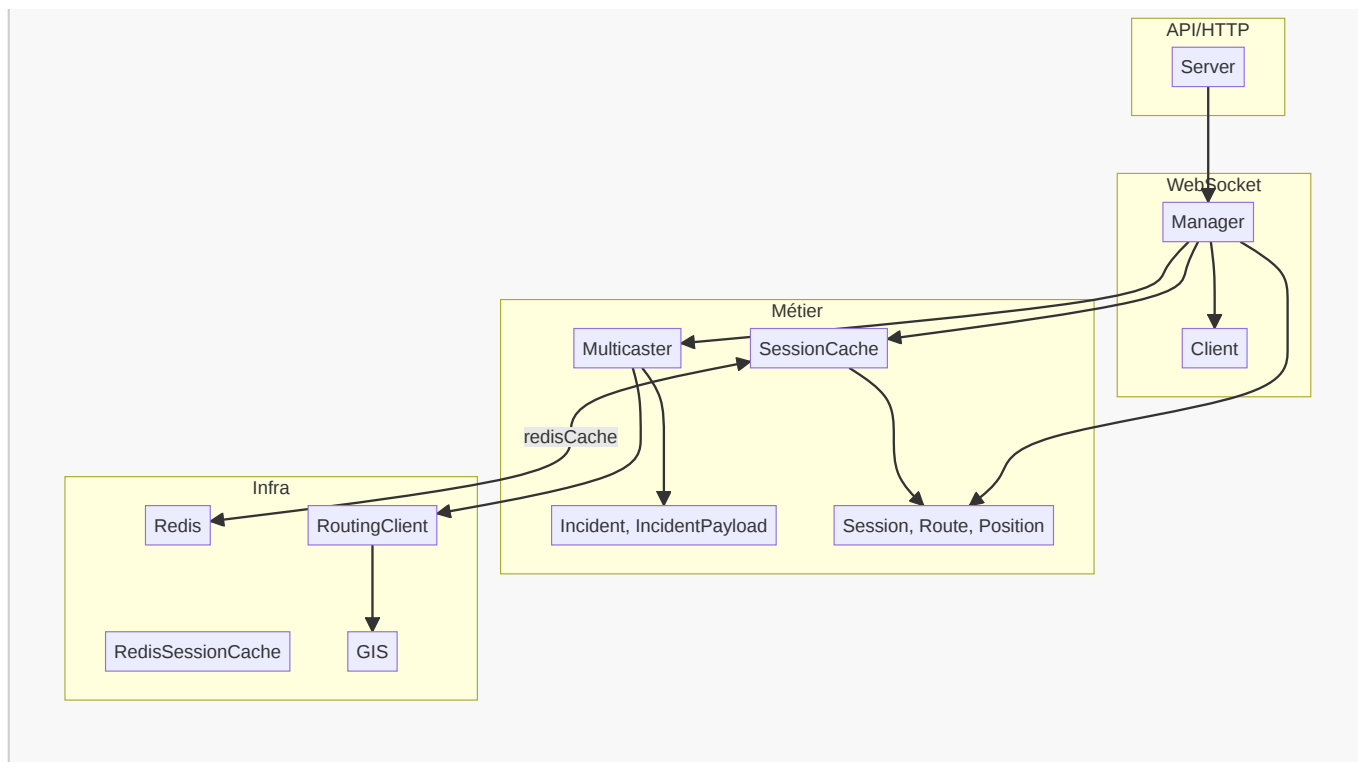
```

- **Usage** : service qui détermine les clients impactés par un incident et les notifie (voire déclenche un recalcul).

7.3. Autres structures clés

- **Server** ([internal/api/server.go](#)) : struct qui encapsule la config, le manager WebSocket et le logger pour le serveur HTTP.
- **RedisSessionCache** ([internal/cache/redis.go](#)) : implémentation concrète de [SessionCache](#) via Redis.
- **Point** ([internal/gis/polyline.go](#)) : structure géographique pour les calculs de distances/incidents.

7.4. Diagramme de dépendances principales



8. Stack trace & appels typiques

8.1. Stack trace d'un flux complet (messages WebSocket)

a) Connexion et initialisation

1. **Client** → **Serveur** : Ouverture WebSocket (`/ws?session_id=...`)
2. **internal/api/handler.go**
 - `wsHandler()` (HTTP → WS upgrade)
 - Appelle `ws.Manager.HandleNewConnection(sessionID, conn)`
3. **internal/ws/manager.go**
 - `HandleNewConnection(id, conn)` → crée un `ws.Client` et lance `Start()`
4. **internal/ws/client.go**
 - `Client.Start()` → lance `readPump()` et `writePump()`
5. **Client** → **Serveur** : Envoi du message `"init"`
6. **internal/ws/client.go**
 - `handleMessage(msg) : case "init"`
 - Désérialise `navigation.Session` et vérifie l'ID
 - Appelle `SessionCache.SetSession(ctx, &session)`

b) Mise à jour de position

1. **Client** → **Serveur** : Message `"position"`
2. **internal/ws/client.go**
 - `handleMessage(msg) : case "position"`
 - Appelle `SessionCache.GetSession(ctx, sessionID)`
 - Met à jour la position dans la session
 - Appelle `SessionCache.SetSession(ctx, session)`

c) Réception d'un incident

1. Incident Pub/Sub (supmap-incidents → Redis)
2. `internal/subscriber/subscriber.go`
 - `Subscriber.Start(ctx)` reçoit le message
 - Appelle `Multicaster.MulticastIncident(ctx, incident, action)`
3. `internal/incidents/multicaster.go`
 - `Multicaster.MulticastIncident()` : boucle sur les clients WebSocket concernés
 - Si besoin, appelle `handleRouteRecalculation()` (recalcul GIS)
 - Appelle `sendIncident() → Client.Send()` (message "incident")
4. `internal/ws/client.go`
 - `writePump()` envoie le message "incident" au client

d) Recalcul d'itinéraire (sur incident bloquant certifié)

1. `internal/incidents/multicaster.go`
 - `handleRouteRecalculation()` :
 - Construit une requête GIS
 - Appelle `routing.Client.CalculateRoute(ctx, req)`
 - Met à jour la session (nouvelle route)
 - Appelle `Client.Send()` (message "route")

8.2. Signatures des fonctions principales impliquées

```
// internal/api/handler.go
func (s *Server) wsHandler() http.HandlerFunc

// internal/ws/manager.go
func (m *Manager) HandleNewConnection(id string, conn *websocket.Conn)
func (m *Manager) Start()

// internal/ws/client.go
func (c *Client) Start()
func (c *Client) handleMessage(msg Message)
func (c *Client) Send(msg Message)
func (c *Client) readPump()
func (c *Client) writePump()

// internal/navigation/session.go
func (r *RedisSessionCache) SetSession(ctx context.Context, session
*navigation.Session) error
func (r *RedisSessionCache) GetSession(ctx context.Context, sessionID
string) (*navigation.Session, error)

// internal/subscriber/subscriber.go
func (s *Subscriber) Start(ctx context.Context) error

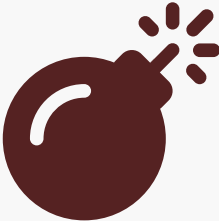
// internal/incidents/multicaster.go
func (m *Multicaster) MulticastIncident(ctx context.Context, incident
```

```
*Incident, action string)
func (m *Multicaster) handleRouteRecalculation(ctx context.Context, client
*ws.Client, session *navigation.Session)
func (m *Multicaster) sendIncident(client *ws.Client, incident *Incident,
action string)
```

8.3. Schéma de séquence illustratif

8.3.1. Flux complet : init, position, incident, recalcul

Ce schéma illustre le parcours d'un message WebSocket typique, depuis la connexion jusqu'à la gestion des incidents et le recalcul d'itinéraire, en montrant chaque acteur et fonction clef impliquée dans le flux.



Syntax error in text

mermaid version 10.4.0

9. Configuration

9.1. Variables d'environnement

Le service utilise un ensemble de variables d'environnement pour sa configuration, chargées automatiquement au démarrage. Voici le tableau récapitulatif (voir aussi [internal/config/config.go](#)) :

Nom	Obligatoire	Description
API_SERVER_HOST	Oui	Hôte d'écoute du serveur HTTP/WebSocket
API_SERVER_PORT	Oui	Port d'écoute du serveur HTTP/WebSocket
REDIS_HOST	Oui	Hôte Redis (cache sessions/navigation)
REDIS_PORT	Oui	Port Redis
REDIS_INCIDENTS_CHANNEL	Oui	Nom du channel Redis Pub/Sub pour les incidents
SUPMAP_GIS_HOST	Oui	Host du service supmap-gis (recalcul d'itinéraire)
SUPMAP_GIS_PORT	Oui	Port du service supmap-gis
ENV	Non	Environnement d'exécution (prod / dev)

9.1.1 Exemple de fichier .env

```
API_SERVER_HOST=0.0.0.0
API_SERVER_PORT=8080
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_INCIDENTS_CHANNEL=incidents
SUPMAP_GIS_HOST=supmap-gis
SUPMAP_GIS_PORT=8000
ENV=dev
```

9.2. CI : build & push automatique (GitHub Actions)

Le dépôt embarque un workflow CI/CD (.github/workflows/image-publish.yml) qui :

- **Déclencheur** : sur chaque push sur la branche **master**
- **Actions** :
 1. Checkout du code
 2. Login au registre de conteneurs GitHub (**ghcr.io**) via **GITHUB_TOKEN**
 3. Build de l'image Docker (taggée **ghcr.io/4proj-le-projet-d-une-vie/supmap-navigation:latest**)
 4. Push de l'image sur GitHub Container Registry

Application mobile (front)

Choix des technologies

Les technologies choisies pour l'application Supmap sont :

- React Native, framework Javascript adapté au développement mobile
- Expo, plateforme servant au développement, build et déploiement de l'application
- Axios, librairie NodeJS pour faire les appels d'API
- React Native Map, librairie permettant l'affichage et manipulation de la carte

Structure du projet

```
supmap-front/
├─ app
│   └─ index.tsx      # Englobe toute l'application
├─ assets/            # Contient toutes les polices d'écriture et images
dont l'application a besoin
├─ components/        # Contient les composants servant à construire
l'application
├─ constants/         # Données constantes servant à l'affichage
├─ contexts/
│   └─ AuthContext.tsx # Contient toute la logique d'authentification de
React Native
├─ navigation/
│   └─ Navigation.tsx  # Enregistre les différents écrans pour permettre
de naviguer entre eux
```

```
├─ screens/           # Contient les écrans navigables et construits à
partir de composants
├─ services/          # Contient une partie de la logique pour alléger
les écrans et composants
└─ README.md
```

Authentification

L'authentification de l'utilisateur est gérée par deux écrans et 3 routes.

- LoginScreen, qui gère la connexion
- RegisterScreen, l'inscription
- ProfileScreen, qui affiche les données utilisateurs et possède un bouton pour se déconnecter

React Native possède également ce qu'il appelle un "AuthContext", possédant une classe User personnalisable et des fonctions permettant de gérer l'utilisateur dans l'application : login(), permettant de transmettre l'information qu'un utilisateur est connecté logout(), permettant de transmettre l'information que l'utilisateur s'est déconnecté

Ces deux fonctions sont importantes car elles définissent l'état de la variable isAuthenticated, utilisé comme son nom l'indique pour savoir si un utilisateur est connecté ou non sur l'application, et donc savoir quel comportement avoir.

```
//Login
let request = {password: password};
request[email.startsWith('@') ? 'handle' : 'email'] = email;
ApiService.post('/login', request).then(async (response) => {
  await saveTokens(response.access_token, response.refresh_token);
  login({ email, handle: email.split('@')[0] });
  navigation.navigate('Home');
})

//Register
ApiService.post('/register', {
  email: email,
  handle: handle,
  password: password
}).then(async (response) => {
  await saveTokens(response.tokens.access_token,
response.tokens.refresh_token);
  login({ email, handle: email.split('@')[0] });
  navigation.navigate('Home');
})

//Logout
logout()
ApiService.post('/logout', {token: getRefreshToken()}).then(async () => {
  await clearTokens()
  navigation.navigate('Home');
})
```


Afin de garder le token d'authentification retourné par les routes login et register, nous utilisons la librairie Expo Secure Storage, permettant de stocker des données localement et de manière sécurisée. Les fonctions nécessaires à l'authentification sont regroupées dans un service AuthStorage.

```
import * as SecureStore from 'expo-secure-store';

const ACCESS_TOKEN_KEY = 'access_token';
const REFRESH_TOKEN_KEY = 'refresh_token';

export const saveTokens = async (accessToken: string, refreshToken?: string) => {
  await SecureStore.setItemAsync(ACCESS_TOKEN_KEY, accessToken);
  if (refreshToken) {
    await SecureStore.setItemAsync(REFRESH_TOKEN_KEY, refreshToken);
  }
};

export const getAccessToken = async (): Promise<string | null> => {
  return await SecureStore.getItemAsync(ACCESS_TOKEN_KEY);
};

export const getRefreshToken = async (): Promise<string | null> => {
  return await SecureStore.getItemAsync(REFRESH_TOKEN_KEY);
};

export const clearTokens = async () => {
  await SecureStore.deleteItemAsync(ACCESS_TOKEN_KEY);
  await SecureStore.deleteItemAsync(REFRESH_TOKEN_KEY);
};
```

Communication avec l'API

Pour communiquer avec l'API nous utilisons la librairie Axios, librairie Node JS permettant de faire des appels aux APIs HTTP.

```
import axios, { AxiosError, AxiosRequestConfig } from 'axios';
import { getAccessToken, getRefreshToken, saveTokens } from './AuthStorage';

const API_BASE_URL = process.env.EXPO_PUBLIC_API_URL;

const api = axios.create({
  baseURL: API_BASE_URL,
  timeout: 10000,
});

const ApiService = {
  get: async (route: string, params?: any) => {
    const response = await api.get(route, { params });
    return response.data;
  },
};
```

```

    post: async (route: string, data?: any) => {
      const response = await api.post(route, data);
      return response.data;
    },

    patch: async (route: string, data?: any) => {
      const response = await api.patch(route, data);
      return response.data;
    },

    delete: async (route: string) => {
      const response = await api.delete(route);
      return response.data;
    },
  };

  export default ApiService;

```

Pour que le code soit plus clair dans les composants, un fichier ApiService se chargera de toute la logique propre aux appels d'API. C'est-à-dire faire l'appel en lui-même, gérer les erreurs, régénérer un access token si celui actuel est expiré.

Le code ci-dessous permet d'intercepter les erreurs renvoyées par l'API, et si celle-ci correspond à un code 401, qui correspond à un token périmé, elle va faire un appel à la route "/refresh" pour en récupérer un nouveau, et exécuter l'appel ayant échoué.

```

api.interceptors.request.use(
  async (config) => {
    const token = await getAccessToken();
    if (token) {
      config.headers.Authorization = `Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3NDY2OTY5NzksIm1hdCI6MTc0Nj
YxMDU3OSwicm9sZSI6IlJPTeVfVWVNFUiIsInVzZXJJZCI6MX0.eyJ1BE_BcabWmj4cC_CZhxcyK
rqPYegd5HemQrvGBt0`;
    }
    return config;
  },
  (error) => Promise.reject(error)
);

api.interceptors.response.use(
  (response) => response,
  async (error: AxiosError) => {
    const originalRequest = error.config as AxiosRequestConfig & {
      _retry?: boolean;
    };
    if (error.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true;
      try {
        const refreshToken = await getRefreshToken();
        const refreshResponse = await

```

```
axios.post(`${API_BASE_URL}/refresh`, {
  refresh_token: refreshToken,
});

const { access_token, refresh_token } =
refreshResponse.data;

await saveTokens(access_token, refresh_token);

originalRequest.headers = {
  ...originalRequest.headers,
  Authorization: `Bearer ${access_token}`,
};

return api(originalRequest);
} catch (refreshError) {
  console.error('Refresh token failed:', refreshError);
  throw refreshError;
}
}

return Promise.reject(error);
};
```



Syntax error in text

mermaid version 10.4.0

Structure des écrans et composants

Les écrans et composants sont ce qui servent à définir l'apparence de l'application. Ils sont structurés de cette manière :

- Props, des variables qu'un component ou écran parent/précédent peut donner à celui-ci pour y avoir accès, exemple :

```
interface Props {
  instruction: any | null;
}

const RouteInstructions: React.FC<Props> = ({ instruction }) => {
  // code
}
```

- Variable/useState, les variables en React se définissent comme un état et se servent d'une méthode associée pour être modifiées, exemple :

```
const [location, setLocation] = useState(null);
const [region, setRegion] = useState(null);
```

- Fonctions, les fonctions sont définies comme des constantes ayant pour valeur une fonction, exemple :

```
const findClosestPolylineIndex = () => {
  let minDistance = Infinity;
  let closestIndex = 0;
  polyline.forEach((point: any, index: any) => {
    const distance = getDistance(
      { latitude: location.latitude, longitude: location.longitude },
      { latitude: point.latitude, longitude: point.longitude }
    );
    if (distance < minDistance) {
      minDistance = distance;
      closestIndex = index;
    }
  });

  return closestIndex;
}
```

- "Retour", le retour de la fonction définissant l'écran ou composant est la partie où on va former sa structure grâce à des balises semblables à du HTML, on utilise des accolades ({}) pour insérer du code dans les balises, exemple complet :

```
<View style={styles.container}>
  <View style={{display: 'flex', flexDirection: 'row', justifyContent:
'space-between'}}>
    <Text style={styles.instruction}>{instruction.street_names.length ?
instruction.street_names.join(', ') : instruction.instruction}</Text>
    <View>
      <MaterialIcons style={{marginBottom: -50}} name=
{instructionsIcons[instruction.type]} size={70} color={'white'}/>
      <Text style={styles.instructionDistance}> {instruction.distanceTo <
1000 ? instruction.distanceTo.toFixed(0) + 'm' :
(instruction.distanceTo/1000).toFixed(2) + 'km'}</Text>
    </View>
  </View>
  <Text style={styles.informationText}>
    {instruction.arrivalTime}
    &nbsp; &#11044; &nbsp; &nbsp;
    {(instruction.remainingDistance / 1000).toFixed(0)} km
    &nbsp; &#11044; &nbsp; &nbsp;
    {(instruction.remainingDuration < 3600 ?
```

```
instruction.remainingDuration / 60 : instruction.remainingDuration /
3600).toFixed(0))} min
</Text>
</View>
```

- Styles, le style de l'écran ou composants se définit en dehors du code de celui-ci, exemple :

```
const RouteInstructions: React.FC<Props> = ({ instruction }) => {
  //code
};

const styles = StyleSheet.create({
  container: {
    position: 'absolute',
    backgroundColor: 'rgba(87,69,138, 1)',
    padding: 15,
    minHeight: 130,
    width: '100%',
  },
});
```

Carte

Pour afficher la carte et tout ce qui y est lié, c'est à dire les markers d'incidents, départ et arrivé, ainsi que le tracé du trajet, nous nous servons de la librairie React Native Map.

La carte se définit comme ceci dans le code :

```
<MapView
  customMapStyle={mapDesign}
  ref={mapRef}
  style={styles.map}
  initialRegion={region}
  showsUserLocation
/>
```

- customMapStyle permet d'appliquer un design personnalisé à la carte
- ref permet d'appliquer plusieurs paramètres à la carte, notamment le fait de suivre l'utilisateur lorsqu'il se déplace
- initialRegion sert à définir la position de base sur la carte au lancement de l'application
- showUserLocation sert à faire ce que son nom indique, soit afficher la position de l'utilisateur

Les Polylines, ou tracés, se définissent comme ceci :

```
<Polyline coordinates={polyline} strokeWidth={5} strokeColor="blue" />
```

- `coordinates` attend un tableau d'objets sous la forme ci-dessous, à partir de ceci la ligne se dessine sur la carte

```
{"latitude": number, "longitude": number}
```

- `strokeWidth` permet de définir la largeur de la ligne
- `strokeColor` permet de définir la couleur de la ligne

Les Markers, ou points sur la carte se définissent comme ceci :

```
<Marker
  key={index}
  coordinate={{ latitude: location.lat, longitude: location.lon }}
  title={location.name ? location.name : `Étape ${index + 1}`}
  pinColor={index === 0 ? 'green' : index ===
route.params.selectedRoute.locations.length - 1 ? 'red' : 'blue'}
/>
```

- `key` sert à donner un identifiant au marker
- `coordinate` sert à définir l'emplacement du marker
- `title` sert à définir quel texte sera afficher lorsque l'utilisateur cliquera sur le marker
- `pinColor` définit la couleur du marker

Calcul de l'itinéraire

Voici le process complet pour calculer l'itinéraire et l'afficher sur l'application :

- L'utilisateur tape sa recherche dans une barre de recherche, ce qui va faire un appel au service `supmap gis` qui va lui même renvoyer des données GPS trouvées par Nominatim.

```
<View style={styles.searchContainer} pointerEvents="box-none">
  {showResults && searchResults.length > 0 && (
    <SearchResultsList searchResults={searchResults} handleClick=
{fetchRoute} />
  )}

  <View style={styles.searchBarContainer}>
    <View style={styles.searchBar}>
      <Ionicons name="search" size={20} color="black" />
      <TextInput placeholder="Où allons-nous ?" value={searchText}
        onChangeText={({text}) => {
          setSearchText(text);
          fetchSearchResults(text);
        }}
        style={styles.searchInput}
      />
    </View>
  </View>
```

```

    </View>
  </View>

  const fetchSearchResults = (text: string) => {
    if(text === '') {
      setSearchResults([])
      setShowResults(false);
    } else {
      ApiService.get('/geocode', {address: text}).then((response) => {
        setSearchResults(response.data)
        setShowResults(true);
      })
    }
  }
}

```

- Les résultats s'affichent dans une liste, ce qui donne la possibilité à l'utilisateur de cliquer sur un des résultats proposés pour faire un appel d'API encore une fois à supmap gis avec les coordonnées GPS nécessaires pour former un trajet, cette fois-ci fourni par Valhalla.

Choix destination :

```

<View style={styles.resultContainer}>
  <FlatList
    data={searchResults}
    keyExtractor={({item, index}) => item.place_id?.toString() ??
index.toString()}
    renderItem={({ item }) => (
      <TouchableOpacity onPress={() => handleResultPress(item)}
style={styles.resultItem}>
        <Text>{item.display_name}</Text>
      </TouchableOpacity>
    )}
  />
</View>

const handleResultPress = async (item: any) => {
  Keyboard.dismiss();
  const destination = {
    lat: parseFloat(item.lat),
    lon: parseFloat(item.lon),
    name: item.display_name,
    type: 'break'
  };
  await fetchRoute([destination], item.display_name);
};

```

Requête à supmap gis et navigation sur la page de choix d'itinéraires, trois au maximum :

```

const fetchRoute = async (destination: any, displayName: string|null) => {
  if (!location) return;
  if (displayName) setSearchText(displayName)
  let points = [{lat: location.latitude, lon: location.longitude, name:
'Votre position', type: 'break'}]
  points = points.concat(destination);
  let request = {
    costing: "auto",
    costing_options: {
      use_tolls: avoidTolls ? 0 : 1
    },
    locations: points
  }
  setLoading(true);
  try {
    ApiService.post('/route', request).then((response) => {
      navigation.navigate('RouteChoice', {routes: response.data,
searchText: searchText});
    })
  } catch (error) {
    Alert.alert("Erreur lors de la récupération de l'itinéraire", error);
  }
  setLoading(false);
};

```

- L'utilisateur peut choisir parmi 3 itinéraires (maximum) différents, ces itinéraires sont affichés avec le temps du trajet et la distance en kilomètres, afin que l'utilisateur puisse choisir ce qu'il préfère entre peu de distance ou un trajet plus rapide. Une fois que le trajet est choisi, l'application navigue de nouveau sur la page de la carte et affiche les instructions de navigation, avec un léger traitement au niveau des instructions et de la "shape", ou polyline, afin de gérer les cas avec plusieurs destinations, qui sont dans des objets distincts.

```

<View style={styles.container}>
  <Text style={styles.title}>Choisissez votre itinéraire</Text>
  <FlatList
    data={routes}
    keyExtractor={({_, index}) => index.toString()}
    renderItem={({ item, index }) => (
      <TouchableOpacity style={styles.card} onPress={() =>
handleSelect(item)}>
        <Text style={styles.label}>Itinéraire
{item.locations[item.locations.length - 1].name}</Text>
        <Text>Temps : {Math.round(item.summary.time / 60)} min</Text>
        <Text>Distance : {(item.summary.length).toFixed(2)} km</Text>
      </TouchableOpacity>
    )}
  />
  <TouchableOpacity style={styles.cancelButton} onPress={handleCancel}>
    <MaterialIcons name={'cancel'} color={'rgba(87,69,138, 1)'} size=
{30} />

```



```

    </TouchableOpacity>
  </View>

  const handleSelect = (selectedRoute) => {
    let completeShape: any[] = [];
    let completeInstructions: any[] = []
    for(let leg of selectedRoute.legs) {
      completeShape = completeShape.concat(leg.shape);
      completeInstructions = completeInstructions.concat(leg.maneuvers);
    }
    selectedRoute.completeShape = completeShape;
    selectedRoute.completeInstructions = completeInstructions;
    navigation.navigate('Home', { selectedRoute: selectedRoute });
  };

```

- Affichage des instructions sur la page, avec le temps et distance restante s'actualisant en temps réel.

```

<View style={styles.container}>
  <View style={{display: 'flex', flexDirection: 'row', justifyContent:
    'space-between'}}>
    <Text style={styles.instruction}>{instruction.street_names.length ?
    instruction.street_names.join(', ') : instruction.instruction}</Text>
    <View>
      <MaterialIcons style={{marginBottom: -50}} name=
      {instructionsIcons[instruction.type]} size={70} color={'white'}/>
      <Text style={styles.instructionDistance}>
        {instruction.distanceTo < 1000 ? instruction.distanceTo.toFixed(0) + 'm' :
        (instruction.distanceTo/1000).toFixed(2) + 'km'}</Text>
      </View>
    </View>
    <Text style={styles.informationText}>
      {instruction.arrivalTime}
      &nbsp; &#11044; &nbsp; &nbsp;
      {(instruction.remainingDistance / 1000).toFixed(0)} km
      &nbsp; &#11044; &nbsp; &nbsp;
      {(instruction.remainingDuration < 3600 ?
      instruction.remainingDuration / 60 : instruction.remainingDuration /
      3600).toFixed(0)} min
    </Text>
  </View>

```

Gestion des incidents

Voici comment la gestion des incidents est faite dans notre application :

- Dès le lancement, l'application va récupérer les types d'incidents et les incidents eux-mêmes dans un rayon de 500 mètres, puis les afficher sur la map, après cela ils ne seront rafraîchis que toutes les 5 minutes.

```

if(!initialLoaded) {
  ApiService.get('/incidents/types').then(response => {
    setIncidentTypes(response);
    ApiService.get('/incidents', {lat: loc.coords.latitude, lon:
loc.coords.longitude, radius: 500}).then(response => {
      setIncidents(response);
      setInitialLoaded(true);
    })
  })
}

{incidents.map((incident: any) => (
  <Marker
    key={incident.id}
    coordinate={{ latitude: incident.lat, longitude: incident.lon }}
    title={incident.type.name}
  >
    <View style={{backgroundColor:
incidentsDesign[incident.type.id].color, borderRadius: 10, paddingRight: 5,
paddingVertical: 5, height: '100%'}}>
      <Text style={{marginLeft: 5}}>
{incidentsDesign[incident.type.id].icon}</Text>
    </View>
  </Marker>
))}

```

- Lorsque l'utilisateur entrera en navigation, il aura accès à un bouton pour en signaler lui même, ouvrant une liste des types d'incidents recueillis par l'appel d'API

```

{instructions.length > 0 && (
  <TouchableOpacity style={styles.incidentButton} onPress={() =>
setShowIncidentModal(true)}>
    <Image style={{resizeMode: 'stretch', height: 40, width: 40, top: 7,
left: 7}} source={require('../assets/images/incidentAddButton.png')}/>
  </TouchableOpacity>
)}

<Modal visible={showIncidentModal} transparent animationType="slide">
  <View style={styles.modalContainer}>
    <View style={styles.modalContent}>
      <Text style={styles.modalTitle}>Sélectionnez un incident</Text>
      {incidentTypes.map((type) => (
        <TouchableOpacity key={type.id} style={{padding: 12,
backgroundColor: incidentsDesign[type.id].color, marginVertical: 5,
borderRadius: 8}} onPress={() => reportIncident(type)}>
          <Text>{type.name} {incidentsDesign[type.id].icon}</Text>
        </TouchableOpacity>
      ))}
      <Button title="Annuler" onPress={() =>
setShowIncidentModal(false)} />
    </View>
  </Modal>

```

```

    </View>
  </Modal>

  const reportIncident = (incidentType: any) => {
    const request = {
      lat: location.latitude,
      lon: location.longitude,
      type_id: incidentType.id,
    }
    ApiService.post('/incidents', request).then(response => {
      setShowIncidentModal(false);
      setIncidents([...incidents, response]);
    })
  }
}

```

Lorsque l'utilisateur se déplace, l'application vérifie automatiquement s'il se rapproche d'un incident. S'il y en a un à moins de 100 mètres, celle-ci ouvre un message demandant à l'utilisateur de confirmer si l'incident est toujours en cours.

```

//Extrait du code s'exécutant à chaque mouvement de l'utilisateur
for(let i = 0; i < incidents.length; i++) {
  let incidentLatLon = {
    latitude: incidents[i].lat,
    longitude: incidents[i].lon,
  }

  if(getDistance(userPosition, incidentLatLon) < 100 &&
!alreadyVotedIncidentsIds.includes(incidents[i].id)) {
    setAlreadyVotedIncidentsIds([...incidents, incidents[i].id]);
    setApproachingIncident(incidents[i])
  }
}

<Modal visible={polyline && approachingIncident !== null && isAuthenticated
&& userId && approachingIncident.user.id !== userId} transparent
animationType="slide">
  <View style={styles.modalContainer}>
    <View style={styles.modalContent}>
      <Text style={styles.modalTitle}>{approachingIncident?.type.name}
      toujours en cours ?</Text>
      <View style={{display: 'flex', flexDirection: 'row'}}>
        <TouchableOpacity style={[styles.incidentInteractionButton,
{backgroundColor: 'rgba(87,69,138, 1)'}]} onPress={upvoteIncident}>
          <Text style=
{styles.incidentInteractionButtonText}>Oui</Text>
        </TouchableOpacity>
        <View style={{width: 50}}></View>
        <TouchableOpacity style={[styles.incidentInteractionButton,
{backgroundColor: 'grey'}]} onPress={downvoteIncident}>
          <Text style=
{styles.incidentInteractionButtonText}>Non</Text>

```

```
        </TouchableOpacity>
      </View>
    </View>
  </View>
</Modal>

const upvoteIncident = () => {
  ApiService.post('/incidents/interactions', {incident_id:
    approachingIncident.id, is_still_present: true}).finally(() =>
    updateIncidents());
}

const downvoteIncident = () => {
  ApiService.post('/incidents/interactions', {incident_id:
    approachingIncident.id, is_still_present: false}).finally(() =>
    updateIncidents());
}

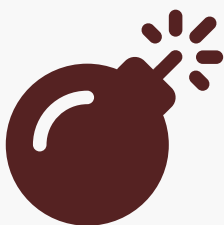
const updateIncidents = () => {
  let updatedIncidents = incidents;
  for (let i = 0; i < updatedIncidents.length; i++) {
    if(updatedIncidents[i].id == approachingIncident.id) {
      updatedIncidents[i].alreadyVoted = true;
      break;
    }
  }
  setApproachingIncident(null);
}
```

Le système de "vote" des incidents permet d'ensuite les certifier afin de les prendre en compte dans le calcul des itinéraires par l'API.

Communication avec les WebSockets

L'application met en place une communication avec un WebSocket afin d'obtenir certaines données en temps réel, notamment les incidents ou le recalcul d'un itinéraire suite à une erreur de l'utilisateur.

Diagramme de séquence sur le comportement de l'application :



Syntax error in text
mermaid version 10.4.0

Mise en place du WebSocket et gestion des réponses selon leurs types, incident pour ajouter un nouvel incident, route pour mettre à jour l'itinéraire :

```

useEffect(() => {
  setUuid().then(() => {
    getItemAsync('websocketUid').then(uuid => {
      ws.current = new WebSocket(API_BASE_URL + "/navigation/ws?
      session_id=" + uuid );

      ws.current.onmessage = (e) => {
        if(e.type == "incident") handleIncidentWebsocket(e.data)
        if(e.type == "route") handleRouteWebsocket(e.data)
      };

      return () => {
        ws.current?.close();
      };
    })
  })
}, []);

```

Envoie de position toutes les 5 secondes :

```

setInterval(() => {
  if (ws.current?.readyState == WebSocket.OPEN && route && route.params
  && route.params.selectedRoute && location && location.latitude &&
  location.longitude) {
    let message = {
      type: "position",
      data: {
        "lat": location.latitude,
        "lon": location.longitude,
        "timestamp": Date.now()
      }
    }
    ws.current.send(JSON.stringify(message));
  }
}, 3000)

```

Récupération d'itinéraire par QR Code

L'application permet de scanner un QR Code généré par l'application web, puis ajouter l'itinéraire contenu dans celui-ci à ceux sauvegardés par l'utilisateur.

L'écran contenant le scan de QR Code faisant une navigation sur celui de la carte, avec un paramètre précis exécutant une requête HTTP dès l'ouverture de celui-ci.

```

<CameraView style={styles.camera} onBarcodeScanned={handleBarCodeScanned}
barcodeScannerSettings={{barcodeTypes: ['qr'],}}>
  <TouchableOpacity onPress={goBack} style={styles.goBackButton}>
    <MaterialIcons name={'arrow-back'} size={30} color=

```

```
{'rgba(87,69,138, 1)'}</TouchableOpacity>
  <View style={{position: 'absolute', top: 80, left: '18%', height: 50,
width: 250, backgroundColor: 'white', borderRadius: 20, justifyContent:
'center', alignItems: 'center'}}>
    <Text>Scannez le QR code disponible</Text>
    <Text> sur le site web</Text>
  </View>
  {scanned && (
    <View style={styles.scannedData}>
      <Text>QR Code Scanné: {qrData}</Text>
      <Text style={styles.reset} onPress={() => setScanned(false)}>
        Scanner à nouveau
      </Text>
    </View>
  )}
  {isDetecting && (
    <View style={styles.qrDetected}>
      <Text style={styles.qrText}>QR détecté...</Text>
    </View>
  )}
</CameraView>

const handleBarCodeScanned = ({ type, data }: { type: string, data: string
}) => {
  if (scanned) return;

  setIsDetecting(true);

  if (!isValidCoordinatesArray(data)) {
    Alert.alert('QR Code invalide', 'Les données doivent être un
tableau avec latitude et longitude.');
```

Démarrer et tester l'application

1. Installer les dépendances

```
npm install
```

2. Créer un compte Expo Go sur <https://expo.dev/signup>

3. Se connecter en utilisant la commande

```
npx expo login -u YOUR_USERNAME -p YOUR_PASSWORD
```

4. Démarrer le serveur expo

```
npx expo start -c
```

Une fois le serveur démarré vous allez voir un QR Code avec une adresse IP en dessous, copiez la

```
> Metro waiting on exp://*votre-ip*:8081
```

5. Fournir les variables d'environnement

L'application n'a besoin que d'une seule variable d'environnement, le fichier .env est donc sous cette forme :

```
EXPO_PUBLIC_API_URL=http://*votre ip*:9000  
// Port 9000 étant celui de la gateway
```

6. Redémarrer le serveur expo pour qu'il prenne bien en compte la modification d'environnement, toujours avec la commande

```
npx expo start -c
```

7. Utilisez une de ces options pour lancer l'application Pour tester l'application voici les différentes options :

- [Émulateur Android](#)
- [Simulateur iOS \(seulement sur Mac\)](#)
- [Expo Go](#), application mobile servant à tester l'application directement sur votre appareil en scannant le QR code donné par la commande "npx expo start", veillez à bien être connecté au même réseau que l'ordinateur hébergeant le serveur et posséder l'application supportant le SDK 52 (Le lien de téléchargement fourni dirige vers une version Android supportant le SDK 52)

Nous recommandons d'utiliser Expo Go dans une environnement de développement.

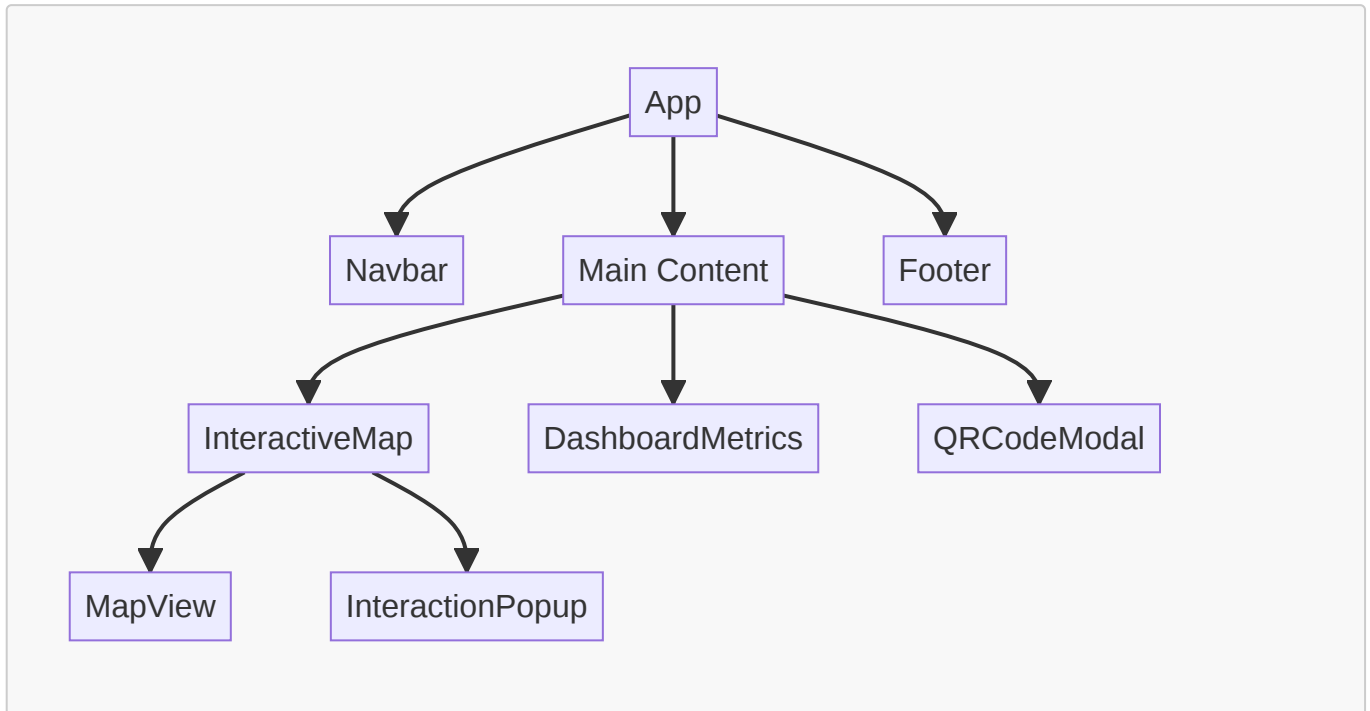
WebApp

Vue d'ensemble

Supmap est une application web de cartographie interactive permettant aux utilisateurs de visualiser des incidents, créer des itinéraires et partager des trajets via QR code.

Architecture des Composants

L'application web est composée d'une seule page. Tous ses éléments sont imbriqués les uns dans les autres selon cette architecture :



Technologies Utilisées

- **Frontend:** React (v19)
- **Cartographie:** Leaflet avec React-Leaflet
- **Styling:** TailwindCSS
- **État Global:** Zustand
- **Requêtes API:** Axios, TanStack Query
- **Build Tool:** Vite
- **QR Code:** react-qr-code

Structure du Projet

Composants Principaux

1. App (**src/App.tsx**)

- Composant racine de l'application
- Gère l'état global des incidents, points utilisateur et formes
- Coordonne les interactions entre les composants

2. InteractiveMap (**src/components/map/InteractiveMap.tsx**)

- Gère la carte interactive
- Fonctionnalités :
 - Affichage des incidents
 - Création d'itinéraires

- Gestion des points utilisateur
- Calcul de trajets

3. DashboardMetrics ([src/components/DashboardMetrics.tsx](#))

- Affiche les métriques et statistiques
- Fonctionnalités :
 - Nombre d'incidents
 - Détails de l'itinéraire
 - Actions rapides (effacer l'itinéraire, générer QR)

Composants Auxiliaires

4. MapView ([src/components/map/MapView.tsx](#))

- Rendu de la carte Leaflet
- Gestion des événements de carte
- Affichage des marqueurs et polygones

5. QRCodeModal ([src/components/QRCodeModal.tsx](#))

- Modal pour générer des QR codes d'itinéraires
- Validation du nom d'itinéraire
- Génération du QR code

Fonctionnalités Clés

1. Gestion des Incidents

- Affichage des incidents sur la carte
- Filtrage par zone géographique
- Popup d'information détaillée
- Mise à jour en temps réel

2. Système de Routage

- Création d'itinéraires personnalisés
- Calcul automatique des trajets
- Affichage des distances et temps estimés
- Points de passage multiples

3. Partage d'Itinéraires

- Génération de QR codes
- Sauvegarde des informations de route
- Interface modale dédiée

4. Interface Utilisateur

- Design responsive
- Navigation intuitive
- Métriques en temps réel
- Thème personnalisé

5. Authentification

- Gestion des tokens JWT
- Refresh token automatique
- Interception des requêtes Axios

Configuration et Environnement

Variables d'Environnement

Les variables d'environnement pour une application frontend avec Vite sont complexes à gérer. Les variables d'environnement sont injectés dans le code transpilé durant le build. Dans notre cas, la seule variable d'environnement nécessaire est l'URL vers la gateway qui expose les services. Nous avons jugé que cette variable n'est en aucun cas critique et avons pris la décision de définir la valeur de cette variable en dur dans le fichier d'environnement Typescript de l'application.

```
// src/config/env.ts
export const env = {
  gatewayHost: "http://localhost:9000",
};
```

DevOps

Authentification

NB: Nous avons spécialement ouvert notre registre d'images au public pour le rendu de ce projet. Il n'est plus nécessaire de s'authentifier pour les télécharger.

Pour déployer les conteneurs constituant le backend, il faut être authentifié par **docker login**.

- Générer un **Personal Access Token** sur GitHub :
 - Se rendre sur <https://github.com/settings/tokens>
 - Cliquer sur **Generate new token**
 - Cocher au minimum la permission **read:packages**
 - Copier le token
- Connecter Docker à GHCR avec le token :

```
echo 'YOUR_GITHUB_TOKEN' | docker login ghcr.io -u YOUR_GITHUB_USERNAME --password-stdin
```

Lancement du projet (complet)

Pour lancer le projet, il vous faudra écrire un fichier `.env` à la racine de `supmap-devops` et contenant les variables d'environnement nécessaire au bon fonctionnement du projet. Un exemple complet et fonctionnel pour un environnement local est disponible dans [.env.example](#).

Sélection de la région pour les services de routing et géocoding

Prenez soin de sélectionner la région qui vous convient le mieux pour vos tests ! Pour ce faire, définissez les variables d'environnements `PBF_NAME` et `PBF_URL` en choisissant le fichier `.pbf` correspondant avec [ce lien](#) Attention à ne pas prendre une région trop grande ! Par exemple la france entière met plusieurs heures à être indexer par le projet. Pendant ce temps, le projet n'est pas utilisable. Par exemple, pour la Basse-Normandie les variables d'environnement seront :

- `PBF_NAME=calvados.osm.pbf`
- `PBF_URL=https://download.openstreetmap.fr/extracts/europe/france/basse_normandie/${PBF_NAME}`

Lancement avec Docker Compose

Placez-vous à la racine du projet `supmap-devops` et exécutez la commande suivante :

```
docker compose up -d
```

NB: Le déploiement est très long, car Valhalla et Nominatim ont besoin d'indexer le fichier téléchargé par le conteneur `supmap-pbf-downloader`. Un health check est configuré sur ces deux services. Lorsque leur état est `Healthy`, alors le projet est totalement opérationnel.

Variables d'environnement

Variable	Type	Description
NOMINATIM_HOST	string	Nom d'hôte du service Nominatim (géocodage)
NOMINATIM_PORT	number	Port du service Nominatim
VALHALLA_HOST	string	Nom d'hôte du service Valhalla (routing)
VALHALLA_PORT	number	Port du service Valhalla
GIS_BASE_DIR	string	Chemin du répertoire contenant les fichiers GIS
PBF_NAME	string	Nom du fichier OpenStreetMap au format PBF
PBF_URL	string	URL de téléchargement du fichier PBF
REDIS_HOST	string	Nom d'hôte du service Redis
REDIS_PORT	number	Port du service Redis
REDIS_INCIDENTS_CHANNEL	string	Nom du canal Redis pour les incidents
JWT_SECRET	string	Clé secrète pour la génération/vérification des JWT
POSTGRES_USER	string	Nom d'utilisateur PostgreSQL

Variable	Type	Description
POSTGRES_PASSWORD	string	Mot de passe PostgreSQL
POSTGRES_PORT	number	Port PostgreSQL
POSTGRES_DB	string	Nom de la base de données principale
POSTGRES_USERS_DB	string	Nom de la base de données des utilisateurs
POSTGRES_INCIDENTS_DB	string	Nom de la base de données des incidents
USERS_DB_URL	string	URL de connexion à la base de données utilisateurs
INCIDENT_DB_URL	string	URL de connexion à la base de données incidents
CONTAINER_INTERNAL_PORT	number	Port interne des conteneurs
SUPMAP_GATEWAY_PORT	number	Port de la passerelle API
SUPMAP_USERS_HOST	string	Nom d'hôte du service utilisateurs
SUPMAP_USERS_PORT	number	Port du service utilisateurs
SUPMAP_INCIDENTS_HOST	string	Nom d'hôte du service incidents
SUPMAP_INCIDENTS_PORT	number	Port du service incidents
SUPMAP_GIS_HOST	string	Nom d'hôte du service GIS
SUPMAP_GIS_PORT	number	Port du service GIS
SUPMAP_NAVIGATION_HOST	string	Nom d'hôte du service navigation
SUPMAP_NAVIGATION_PORT	number	Port du service navigation

Services externes

Valhalla

Valhalla est un moteur de routage open source qui nous permet de :

- Calculer des itinéraires entre deux points
- Obtenir les instructions de navigation détaillées
- Optimiser les trajets en fonction de différents modes de transport (voiture, vélo, piéton)

Dans notre projet, il est principalement utilisé pour la fonctionnalité de navigation et le calcul d'itinéraires alternatifs en cas d'incidents.

Nominatim

Nominatim est un service de géocodage qui nous permet de :

- Convertir des adresses textuelles en coordonnées géographiques (géocodage)
- Convertir des coordonnées géographiques en adresses (géocodage inverse)
- Rechercher des lieux par nom ou type

Dans notre application, il est utilisé pour la recherche d'adresses et la conversion des coordonnées en adresses lisibles.

Redis / RedisInsight

Redis est une base de données en mémoire que nous utilisons pour :

- La gestion des événements en temps réel via le système pub/sub
- Le partage d'informations sur les incidents en cours
- La mise en cache de certaines données

RedisInsight est une interface graphique web qui nous permet de monitorer et gérer notre instance Redis.

Postgres

PostgreSQL est notre système de gestion de base de données (SGBD) relationnel qui stocke :

- Les données des utilisateurs (authentification, profils)
- Les informations sur les incidents (localisation, type, statut)
- L'historique des incidents

Notre architecture utilise plusieurs bases de données PostgreSQL distinctes pour séparer les différentes préoccupations (utilisateurs et incidents).