Projet GLA : GPS-Like
Conception

Projet GLA encadré par Burkhart Wolff

4 Mars 2019

Groupe 7

Cerveau Eric

Malassé Juliette

Shenjin Lyu

1 Diagramme de classe de conception

Plusieurs corrections on été apportées depuis l'analyze :

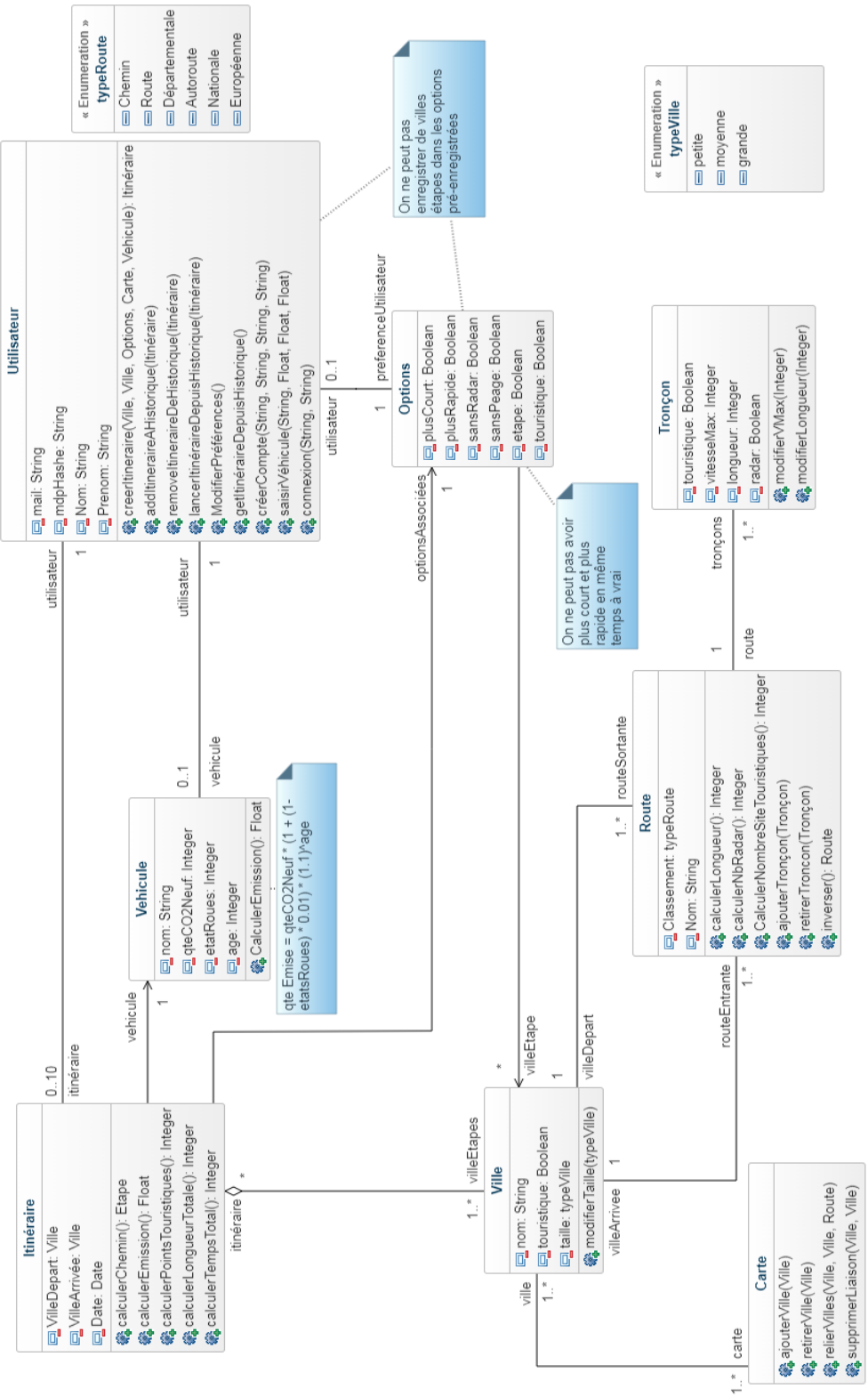
— on a supprimé la classe historique étant donné qu'elle n'apportait rien de plus qu'une simple liste qu'on pouvait enregistré dans la classe d'utilisateur, à laquelle on ajoutait juste des fonction pour gérer cette liste comme par exemple supprimer un élément et en ajouter un.

— on a ajouté un lien entre la classe itinéraire et la casse véhicule car on ne pouvait pas remonter au véhicule utilisépour un itinéraire et on veut pouvoir stocker dans la base de donnée quel véhicule est utilisé pour un itinéraire donné

— on a rajouté des détails quant au champ d'options sauvegardées par l'utilisateur, on ne veut pas que celui-ci pré-enregistre un set d'option avec des villes-étapes obligatoires.

— on a ajouté une classe carte, car on s'est rendu compte que plusieurs cartes pouvaient etre enregistrées par les gestionnaires dans la base de donnée de l'application, mais qu'une seule était active pour les utilisateurs.

On a aussi ajouté les fonctions des différentes classes semblants nécessaires, sans précisés les getters et les setters pour ne pas surcharger le diagramme.


« Enumeration »
typeRoute
Chemin
Route
Départementale
Autoroute
Nationale
Européenne
Utilisateur
mail: String
mdpHashe: String
Nom: String
Prenom: String
creerItineraire(Ville, Ville, Options, Carte, Vehicule): Itinéraire
addItineraireAHistorique(Itinéraire)
removeItineraireDeHistorique(Itinéraire)
lancerItinéraireDepuisHistorique(Itinéraire)
ModifierPréférences()
getItinéraireDepuisHistorique()
créerCompte(String, String, String, String)
saisirVéhicule(String, Float, Float, Float)
connexion(String, String)
On ne peut pas enregistrer de villes étapes dans les options pré-enregistrées
« Enumeration »
typeVille
petite
moyenne
grande
utilisateur
1
0..10
itinéraire
utilisateur
1
0..1
vehicule
1
préferenceUtilisateur
1
0..1
utilisateur
Options
plusCourt: Boolean
plusRapide: Boolean
sansRadar: Boolean
sansPeage: Boolean
etape: Boolean
touristique: Boolean
optionsAssociées
1
Tronçon
touristique: Boolean
vitesseMax: Integer
longueur: Integer
radar: Boolean
modifierVMax(Integer)
modifierLongueur(Integer)
tronçons
1..*
route
1
Vehicule
nom: String
qteCO2Neuf: Integer
etatRoues: Integer
age: Integer
CalculerEmission(): Float
qte Emise = qteCO2Neuf * (1 + (1-etatsRoues) * 0.01) * (1.1)^age
On ne peut pas avoir plus court et plus rapide en même temps à vrai
vehicule
1
Route
Classement: typeRoute
Nom: String
calculerLongueur(): Integer
calculerNbRadar(): Integer
CalculerNombreSiteTouristiques(): Integer
ajouterTronçon(Tronçon)
retirerTroncon(Tronçon)
inverser(): Route
routeSortante
1..*
Itinéraire
VilleDepart: Ville
VilleArrivée: Ville
Date: Date
calculerChemin(): Etape
calculerEmission(): Float
calculerPointsTouristiques(): Integer
calculerLongueurTotale(): Integer
calculerTempsTotal(): Integer
itinéraire
*
villeEtapes
1..*
Ville
nom: String
touristique: Boolean
taille: typeVille
modifierTaille(typeVille)
villeEtape
*
1
villeDepart
1
routeEntrante
1..*
villeArrivee
1
ville
1..*
carte
1..*
Carte
ajouterVille(Ville)
retirerVille(Ville)
relierVilles(Ville, Ville, Route)
supprimerLiaison(Ville, Ville)

2 Technologies utilisées

Souhaitant réaliser une application utilisable sur n'importe quel support, puisque la plupart des smartphones disposent d'un navigateur web, le groupe s'est mis d'accord pour réaliser une application web. Un problème c'est ensuite posé : Juliette n'ayant jamais réalisé d'application web et les deux autres membres du groupe ayant déjà réalisé des projets en utilisant nodeJS, ceux-ci ont proposé au groupe de réaliser une application se basant sur NodeJS, permettant aux membres n'ayant jamais fait de web de n'apprendre principalement qu'un seul langage : le Javascript.

De plus, nodeJS apporte un avantage non négligeable, son gestionnaire de paquets extrèmement bien fourni qui nous permet de trouver des outils déjà fonctionnels.

On réalisera l'application en suivant le Modèle-Vue-Controler (MVC) :

— Un modèle (Model) contient les données à afficher.

— Une vue (View) contient la présentation de l'interface graphique.

— Un contrôleur (Controller) contient la logique concernant les actions effectuées par l'utilisateur.

2.1 NodeJS

Node.js est une plateforme logicielle libre et événementielle en JavaScript orientée vers les applications réseau qui doivent pouvoir monter en charge, elle se base sur l'utilisation du moteur Javascript V8 de Chrome et permet de réaliser des applications web dont le backend est écrit en Javascript.

2.2 NPM

Npm (Node.js package manager) est le gestionnaire de paquet de nodeJS par défaut. C'est l'un des plus utilisés et est facile d'utilisation.

2.3 Express

NodeJS étant par défaut extrêmement bas niveau, et donc particulièrement lourd à utiliser. Express.js est un framework destiné à la réalisation d'applica-

tion web basée sur nodeJS. Il allège considérablement le code nécessaire pour réaliser une application web.

2.4 MongoDB

MongoDB est un gestionnaire de base de données couramment utilisée avec NodeJS. On utilisera mongoose pour faire la liaison entre l'application et la base de données.

2.5 Angular

Angular permet de créer facilement des applications en une seule page. Angular est géré par Google et est mis a jour périodiquement. Angular utilise le typeScript, qui utilise des classes. On utilisera donc Angular pour nos Vues.

De plus, Angular fournit des outils permettant de réaliser facilement des tests unitaires.

3 Calculs des itinéraires

3.1 Dans le cas où l'utilisateur ne cherche pas un chemin touristique

Calculer un itinéraire revient à calculer le plus court chemin sur un graphe : les villes sont des états et on a des transitions qui représentent les routes reliant ces états. En fonction des différentes options choisies par l'utilisateur, on changera le poids/coût des arrêtes :

— Si on souhaite réaliser le chemin le plus rapide, alors on calculera longueur /(vitesseMaximum-5) de chaque tronçon. On retire 5km/h à la vitesse maximale car un conducteur est rarement au maximum de la vitesse autorisée 100% du temps durant un trajet. On additionnera ensuite la valeur obtenue pour chaque tronçon d'une route pour obtenir le temps nécessaire au parcours de cette route.

— Si on souhaite réaliser le chemin le plus court en terme de distance, alors on aura juste à réaliser les mêmes opérations en se basant uniquement sur la longueur.

Pour gérer l'option "pas de péage", on va juste donner un coût infini à l'arrête représentant une route comportant des tronçons payants. De la même manière, on va ainsi éliminer les routes comportant des tronçons avec des radars pour l'option sans radar.

Si l'itinéraire comporte des étapes V_i avec $1 \leq i \leq k$ réalisera k+1 fois l'algorithme de A-STAR :

— du départ à l'étape V_1

— de l'étape V_1 à l'étape V_2

— ...

— de l'étape V_k à l'étape de destination.

Dans le cas où l'utilisateur sortirait de la route, on calcule l'itinéraire position actuelle -> dernière étape proposée puis on reprend l'itinéraire normal. Dans ce cas, l'utilisateur a juste échoué à rejoindre la dernière étape proposée, il semble donc plausible que dans la majorité des cas l'étape proposée ne soit pas loin et on ne rajoutera donc que peu d'étapes intermédiaires.

3.2 Algorithme de A-STAR

Pour ce qui est de l'implémentation, on utilisera une version déjà impllémentée et optimisée disponible sur npm à l'adresse suivante :
https ://www.npmjs.com/package/ngraph.path

Nous détaillerons quand même le fonctionnement de l'algorithme de base, ici dans un exemple simple où l'on cherche à trouver un chemin sur un plateau de jeu représentable par un tableau :

Dans une premier temps on définit la structure de données *Noeud* nécessaire pour le fonctionnement de l'algorithme et un noeud de départ. Valx et valY sont simplement les coordonnées de x et y.
structure Noeud = {
x,y : Nombre
cout, heuristiqu : Nombre
}
depart $\leftarrow Noeud(valX, valY, 0, 0)$

On aura aussi besoin d'une fonction pour comparer deux noeuds. On réalise donc une fonction compare2Noeuds qui permet de savoir si un noeud est plus près qu'un autre du départ.

Algorithme 3.2: COMPARE2NOEUDS($n1 : Noeud, n2 : Noeud$)

si $n1.heuristique < n2.heuristique$
retourne (1)
sinon si $n1.heuristique == n2.heuristique$
retourne (0)
sinon retourne (-1)

Pour finir l'algorithme du plus court chemin :

Algorithme 3.3: A-STAR(g : $graph, objectif$: $Noeud, depart$: $Noeud$)

$closedList \leftarrow File()$
$openList \leftarrow FilePrioritaire(comparateur = compare2Noeuds)$
$openList.ajouter(depart)$
tant que $openList \neq Vide$
{
$u \leftarrow opentList.depiler()$
si $u.x == objectif.xetu.y == objectif.y$
retourne $(reconstituerChemin(u))$
pour tout *voisin v de u dans g*
{
si *v existe dans closedList avec un cout inferieur* **ou**
v existe dans openList avec un cout inferieur
$neRienFaire()$
sinon {
$v.cout \leftarrow u.cout + 1$
$v.heuristique \leftarrow v.cout + distance([v.x, v.y], [objectif.x, objectif.y])$
$openList.ajouter(v)$
}
$closedList.ajouter(u)$
}

3.3 Dans le cas où l'utilisteur cherche un chemin touristique

Pour ce qui est des chemins touristiques, le problème est à prendre dans l'autre sens : dans tous les autres cas, on pensait en terme de cout, on souhait trouvé l'itinéraire le moins long, que ce soit en temps ou en terme de distance, celui où il n'y a pas de radar, de péages. On pouvait toujours chercher les options qui prenaient le moins de quelquechose, et les algorithme pour trouver le plus court chemin sont plutot efficaces pour ce genre de problèmes.

Dans le cas des itinéraires touristiques, le problème se pose de la manière suivante : on ne cherche pas à avoir le moins de quelquechose, mais le plus de points touristiques possibles pour un itinéraire n'étant pas forcément beaucoup plus, et surtout un itinéraire qui ne boucle pas. Car, si on cherche à trouver

un chemin avec le plus de points touristiques possible, on risque fort d'avoir un chemin infini suivant une boucle.

La solution la plus simple a réaliser serait donc de calculer tous les chemins possibles entre le noeud de départ et le noeud cible, sans boucle. Ensuite on calculerait un score de la manière suivante :

— si on veut un chemin court en temps et comportant beaucoup de points touristiques, on calcule chemin.nbPointsTouristiques()/ chemin.calculerTempsTotal(). Le chemin avec le meilleur score sera celui selectionner par l'algorithme.

— la même chose mais avec le ratio chemin.nbPointsTouristiques() /chemin.calculerLongueurTotale() .

On pourra dans le futur rajotuer une fonctionnalité permettant d'ajouter une longueur/une cout en temps qui n'excède pas un certain montant pour éviter de sélectionner des chemins trop longs.

3.4 Algorithme pour trouver tous les chemins possibles

3.4.0.1 recuperer le nombre de villes

int nbVilles

3.4.0.2 Creer une matrice de taille nbVilles x nbVilles

$oncon[][]mt \leftarrow Troncon[nbVilles][nbVilles]$

3.4.0.3 Remplir mt avec null au début

Algorithme 3.5: INITMT($mt : Matrice$)

pour tout i *de* $nbVilles$
{**pour tout** j *de* $nbVilles$
{$mt[i][j] \leftarrow null$

3.4.0.4 Inserer les troncons existants

Algorithme 3.6: REMPLIRTRON($routes : Route[]$)

pour tout i *de* $routes$

$\begin{cases} \textbf{pour tout } j \textit{ de } troncons \\ \begin{cases} Onrcuprelesidentifiantsdej.ville1enidv1etdej.ville2enidv2 \\ mt[idv1][idv2] \leftarrow jmt[idv1][idv2] \leftarrow j \end{cases} \end{cases}$

3.4.0.5 Exemple de matrice obtenue mt (Annexe I)

départ | arrivée

	0	1	2	3	4	5	6	7	8
0	null	null	tron	tron	null	null	null	null	null
1	null	null	null	null	null	tron	tron	null	null
2	tron	null	null	tron	null	null	null	null	tron
3	tron	null	tron	null	tron	tron	null	null	null
4	null	null	null	tron	null	tron	null	null	null
5	null	tron	null	tron	tron	null	null	null	null
6	null	tron	null	null	null	null	null	tron	null
7	null	null	null	null	null	null	tron	null	tron
8	null	null	tron	null	null	null	null	tron	null

0 : Paris, 1 : Orange, 2 : Wissous, 3 : Evry, 4 : Auxerre, 5 : Lyon, 6 : Montpellier, 7 : ceyras, 8 : Clermont-ferrand

tron : il existe un troncon entre ville i et ville j

null : il n'existe pas de troncon

3.4.0.6 Une structure ou une classe de chemin

string [ ] villes

Vector <Troncon> trons

int longueurs

int temps

bool radars

bool peages

int touristiques

3.4.0.7 Enregistrer les chemins dans ce vecteur

Vector <chemin> chemins

3.4.0.8 Parcourir tous les routes avec une fonction récursive

Algorithme 3.8: TROUVECHEMINS($idDepart : ID, idArrivee : ID, lv : int[], vt : VectorTroncon, sommeLqr : int, sommeTemps : int, radars : bool, peages : bool, touristique : int$)

si idDepart == idArrivee
  alors creer chemin avec lv, vt, sommeLgr, sommeTemps, radars, peages, touristiques
chemins ajoute chemin
  sinon lv ajoute idDepart
pour tout i *de* $nbVilles$
si mt[idDepart][i] !=null et i not in lv
  alors vt ajoute mt[idDepart][i]
trouveChemins(i,arrivee,lv,vt,
sommeLgr+mt[idDepart][i].longueur,
sommeTemps+mt[idDepart][i].longueur/mt[idDepart][i].vitesse,
radars||mt[idDepart][i].radar,
payant||mt[idDepart][i].payant,
touristique+mt[idDepart][i].touristique)

3.4.0.9 Trier les chemins par le plus court

Algorithme 3.9: triPlusCourt(*cms* : Vector chemin)

$nb \leftarrow cms.size()$

int pos

chemin cpt

pour tout *i de* 1 *a nb*

$cpt \leftarrow cms.get(i)$

$pos \leftarrow i - 1$

tant que pos >= 0 et cms.get(pos).longueurs > cpt.longueurs

$cms.get(pos + 1) \leftarrow cms.get(pos)$

$pos \leftarrow pos - 1$

$cms.get(pos + 1) \leftarrow cpt$

3.4.0.10 Trier les chemins par le plus rapide

Algorithme 3.10: triPlusRapide(*cms* : Vector chemin)

$nb \leftarrow cms.size()$

int pos

chemin cpt

pour tout *i de* 1 *a nb*

$cpt \leftarrow cms.get(i)$

$pos \leftarrow i - 1$

tant que pos >= 0 et cms.get(pos).temps > cpt.temps

$cms.get(pos + 1) \leftarrow cms.get(pos)$

$pos \leftarrow pos - 1$

$cms.get(pos + 1) \leftarrow cpt$

3.4.0.11 Trier les chemins par le plus touristique

Algorithme 3.11: triPlusTouristiqueEtCourt(*cms* : Vector chemin)

nb ← *cms.size*()
int pos
chemin cpt
pour tout *i de* 1 *a nb*
{
  cpt ← *cms.get*(*i*)
  pos ← *i* − 1
  tant que pos >= 0 et cms.get(pos).touristiques/cms.get(pos).longueurs > cpt.touristique/cpt.longueurs
  {
    cms.get(*pos* + 1) ← *cms.get*(*pos*)
    pos ← *pos* − 1
  }
  cms.get(*pos* + 1) ← *cpt*
}

3.4.0.12 Filter les chemins sans radars

Algorithme 3.12: sansRadars(*cms* : Vector chemin)

nb ← *cms.size*()
Vector <chemin> cheminSr
i ← 0
tant que i < nb
{
  si ! cms.get(i).radars
    alors *cheminsSr.add*(*cms.get*(*i*))
  i + +
}
retourne (*cheminsSr*)

3.4.0.13 Filter les chemins sans péages

Algorithme 3.13: SANSPEAGES(*cms* : Vector chemin)

$nb \leftarrow cms.size()$

Vector <chemin> cheminSp

$i \leftarrow 0$

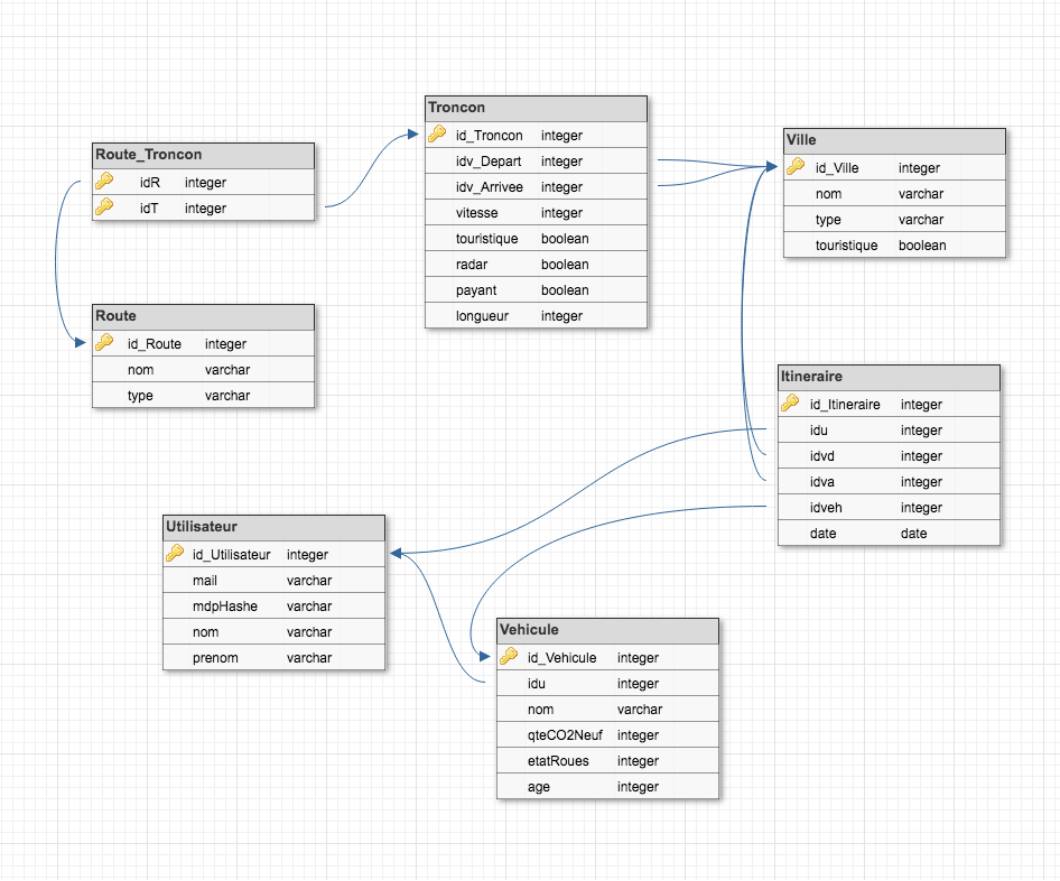
tant que i < nb

- **si** ! cms.get(i).peages
  - **alors** $cheminsSp.add(cms.get(i))$
- $i++$

retourne ($cheminsSp$)

4 Conception de la base de données

4.1 MCD



4.2 Dictionnaire de données

ndlr : tous les champs types marqué "ID" correspondents à des ObjectID dans mongoDB.

Table	Code	Intituieé	type	Longueur	Nature	Règle
Ville	idVille	Identifiant de ville	ID	10	E	
	nom	Nom de ville	varchar	20	E	
	type	Type de ville (petite, moyenne, grande)	Enum		E	
	touris - tique	Ville touristique	bool		E	
Troncon	idTron -con	identifiant de tron-con	ID	10	E	
	idv De-part	Identifiant de ville de départ	ID	10	E	
	idv Arri-vee	Identifiant de ville d'arrivée	ID	10	E	
	vitesse	Vitesse maximum sur un troncon	int	3	E	<=130
	touris - tique	Troncon touris-tique	bool		E	
	radar	Si radar existe dans le troncon	bool		E	
	payant	S'il faut payer pour passer le troncon	bool		E	
	long - ueur	Longueur de tron-con	int	10	E	>=0

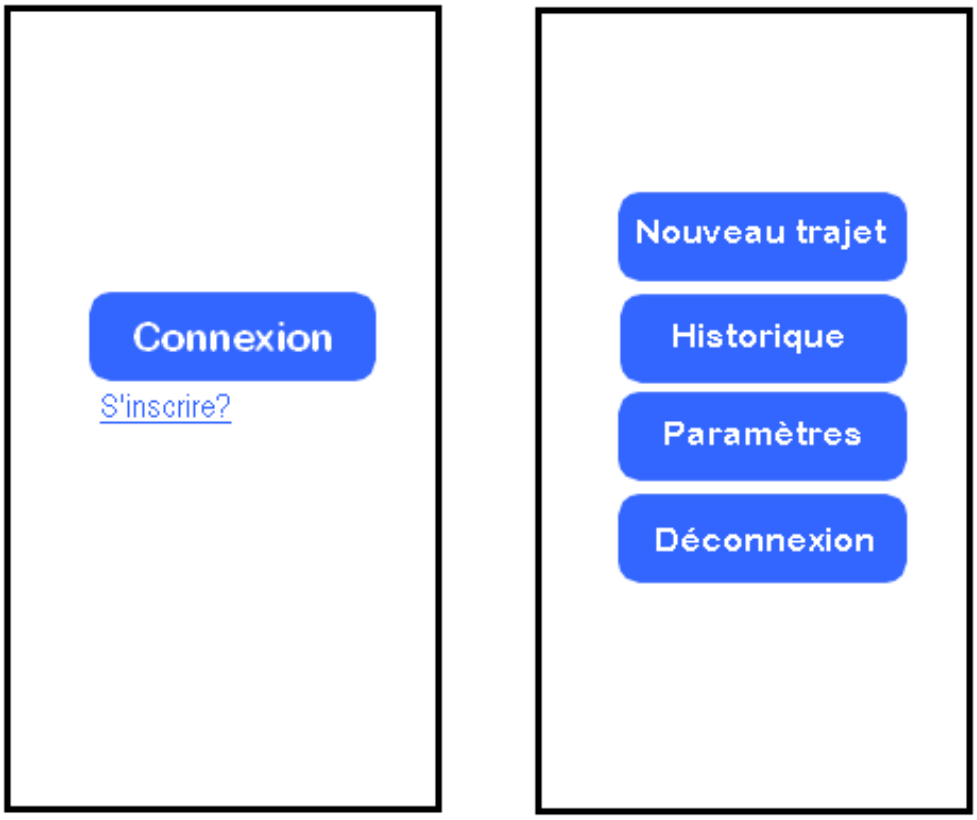
Route	id Route	Identifiant de route	ID	10	E	
	nom	Nom de route	varchar	20	E	
	type	Type de route	Enum		E	
Utilisateur	id Utili - sateur	Identifiant d'utilisateur	ID	15	E	
	mail	Email d'utilisateur	varchar	20	E	
	mdp Hashe	Mot de passe d'utilisateur	varchar	20	E	
	nom	Nom d'utilisateur	varchar	20	E	
	prenom	Prenom d'utilisateur	varchar	20	E	
Vehicule	id Vehicule	Identifiant de véhicule	ID	20	E	
	idu	Identifiant d'utilisateur	ID	15	E	
	nom	Nom de véhicule	varchar	20	E	
	qte CO2 Neuf	Quantité de CO2 de véhicule	int	4	E	
	etat Roues	Etat de roues de véhicule	int	4	E	
	age	Nombre d'années de véhicule	int	4	E	

Iteneraire	id Itene -raire	Identifiant d'itene-raire	ID	10	E	
	idu	Identifiant d'utili-sateur	ID	10	E	
	idvd	Identifiant de ville de départ	ID	10	E	
	idva	Identifiant de ville d'arrivée	ID	10	E	
	idveh	Identifiant de véhi-cule	ID	10	E	
	date	Date d'histoire	Date		E	

5 Conception de l'interface

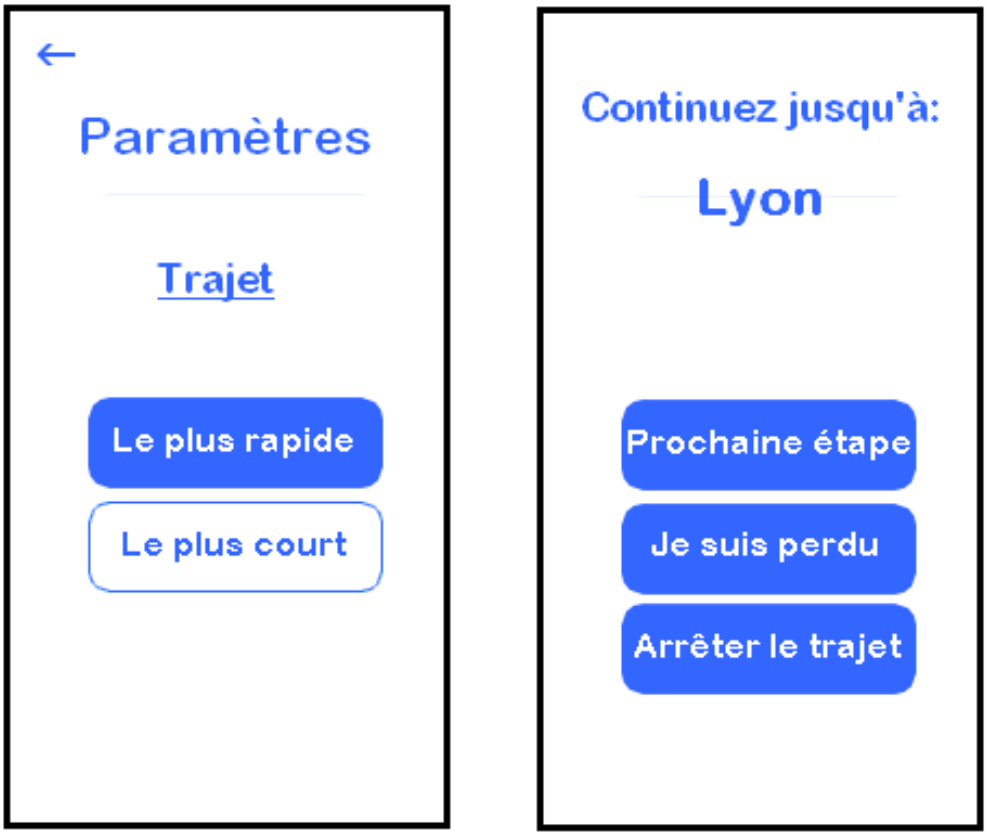
La conception de l'interface est une partie très importante de notre système GPS. En effet, nous avons pour but de délivrer une apllication la plus ergonomique possible. Pour cela il faut peu de boutons à l'écran, des fonctionnalités claires pour chacun et un fonctionnement intuitif.

Sur l'écran d'accueil, on trouvera un bouton de connexion et un lien à cliquer pour s'inscrire qui mènera à un formulaire d'inscription. On aura également l'accueil, avec 4 possibilités : saisir un nouveau trajet, consulter son historique, consulter et modifier les paramètres et se déconnecter.



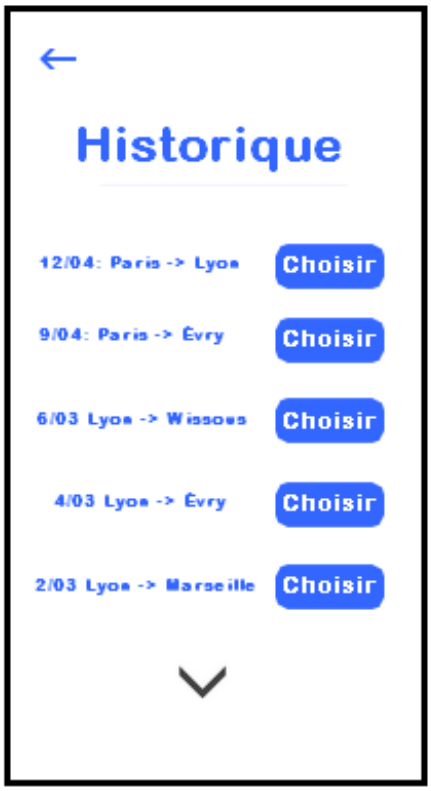
- Interfaces de connexion et accueil -

Les paramètres se feront sous forme de choix, parfois exclusifs. Par exemple, si le bouton le plus rapide est cliqué, on ne pourra pas choisir "le plus court" sans que "le plus rapide" soit désactivé. On pourra également retourner sur des paramètres modifiés précédemment grâce à une flèche de retour. Pour le suivi du trajet, on indique simplement la ville à suivre et on permet à l'utilisateur d'indiquer qu'il est prêt à aller à la prochaine étape, qu'il est perdu ou qu'il veut simplement arrêter le trajet.



- Interface de modification des paramètres et suivi d'un trajet -

Pour l'historique, on affichera seulement 5 résultats à la fois par souci de clarté où seront affichés la date du trajet et les villes de départ et d'arrivée. On pourra retourner au menu ou descendre plus bas dans l'historique grâce à deux flèches. On peut également choisir d'effectuer un trajet enregistré dans l'historique en cliquant sur le bouton "choisir".



- Interface de consultation de l'historique -

6 Diagrammes de séquence

La plupart des diagrammes de séquence ne changent pas tous fondamentalement par rapport à l'analyse, nous avons ajouté ceux qui présentaient un réel changement. La consultation d'historiques ne change pas si ce n'est qu'on va rechercher les éléments de l'historique dans la base de données. Le suivi d'un trajet n'interagit pas avec l'application de manière active, il se contente de suivre les villes-étapes et d'appeler la fonction de redirection en cas d'égarement de l'utilisateur.

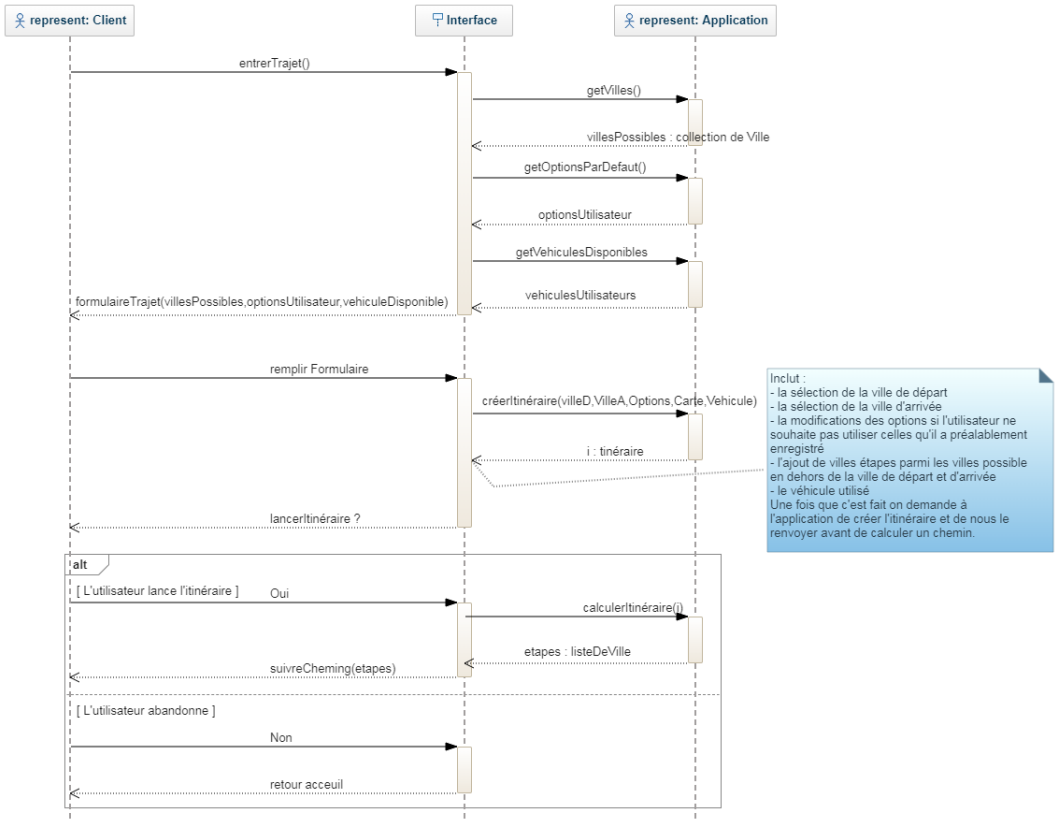
6.1 Spécifier un trajet et le lancer

Pour créer un trajet, un utilisateur doit s'être connecté au préalable. Lorsqu'il souhaite créer un trajet, l'interface va demander à l'application de lui fournir :

— la liste des villes disponibles pour le guidage

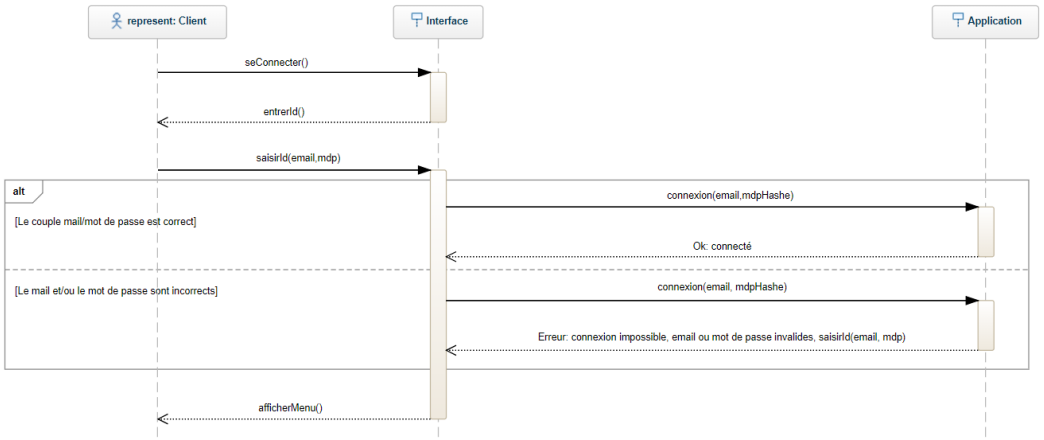
— les options par défaut de l'utilisateur

Puis affiche ensuite un formulaire. A partir de là, l'utilisateur spécifie la ville où il se trouve et la ville où il souhaite se rendre. Il peut aussi modifier les options si ses choix par défaut ne conviennent pas, et peut ajouter des villes étapes s'il le souhaite, on ne peut pas passer plusieurs fois par la même ville, donc chaque fois qu'il souhaitera ajouter une étape on enlèvera les villes par lesquelles il souhaite déjà passer. L'utilisateur sélectionne ensuite le véhicule qui va utiliser pour ce trajet. Une fois que l'itinéraire est enregistrer par l'application, l'interface propose a l'utilisateur de lancer l'itinéraire. Si l'utilisateur veut suivre un itinéraire, on calcule le chemin approprié et on guide l'utilisateur (ce qui sera détailler dans "Suivre un itinéraire") Sinon, l'utilisateur retourne à l'écran d'acceuil.



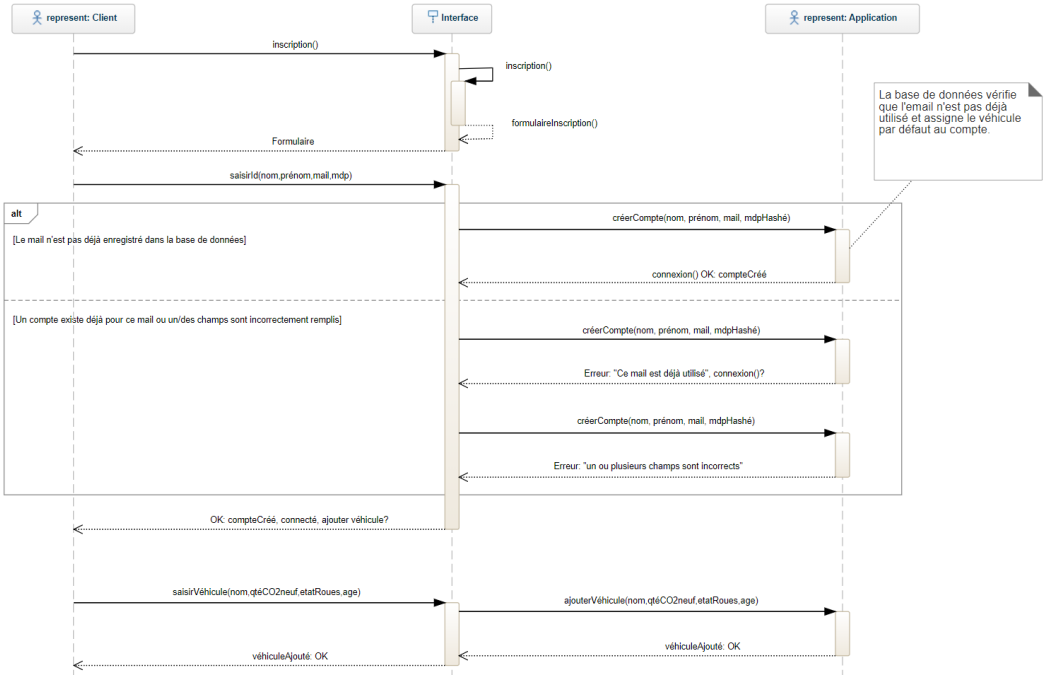
6.2 Se connecter

Pour se connecter, un utilisateur doit spécifier son email et son mot de passe. Ce couple est alors recherché dans la base de données et si l'on trouve un couple correspondant, l'utilisateur est connecté. Sinon, on le redirige vers le formulaire d'inscription si son mail n'est pas dans la base de donnée ou bien on lui propose de retaper son mot de passe si son mail est présent.



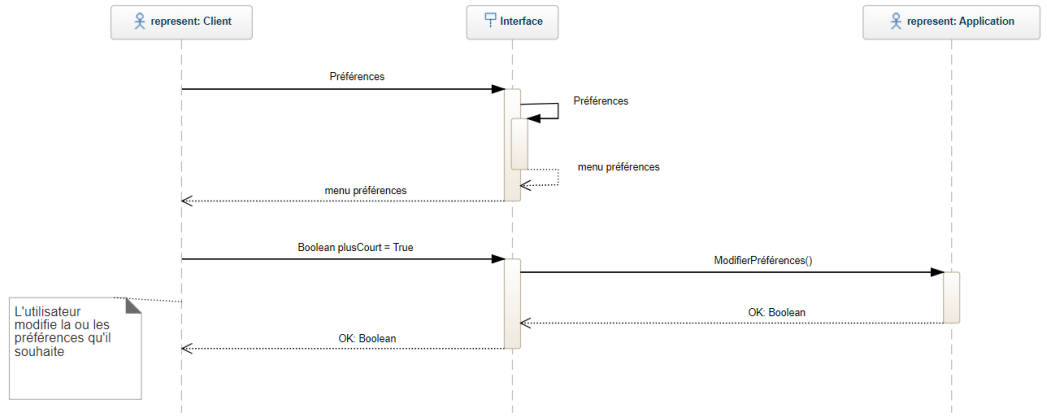
6.3 S'enregistrer

Pour s'enregistrer, l'utilisateur doit indiquer mail, mot de passe, nom et prénom. Le mail doit être unique dans la base de données, sinon on redirige l'utilisateur vers le formulaire de connexion.



6.4 Modifier les préférences

Pour modifier les préférences, l'utilisateur choisit les préférences ce qui met à jour les booléens correspondants à ces préférences, qui sont ensuite considérées comme modifiées.



7 Autres fonctions à détailler

Certaines fonctions étaient plus compliquées à détailler avec un diagramme de séquence plutot qu'en donnant leur pseudocode, vous trouverez donc leur pseudocode dans les parties qui suivent

7.1 Algorithme calcul du nombre de point touristiques d'un itinéraire

On utilisera une fonction récursive pour calculer le nombre de points touristiques sur un itinéraire :

Algorithme 7.1: CALCULERPOINTSTOURISTIQUES(*cible : Itinraire*)

retourne (*nbPointTouristiques*(*cibles.villesEtapes*))

Algorithme 7.2: NBPOINTSTOURISTIQUES(*etapes : ListeDeVille*)

si *etapes.tete*() = *vide*
retourne (0)
si *etapes.tete*().*estTouristique*()
{
pour tout *route qui sort de etapes.tete*()
{**si** *route.villeArrivee* == *next*(*etapes*)
retourne (1 + *nbPointsTouristiques*(*route*) + *nbPointTouristiques*(*next*(*etapes*)))
}
sinon
{
pour tout *route qui sort de etapes.tete*()
{**si** *route.villeArrivee* == *next*(*etapes*)
retourne (*nbPointsTouristiques*(*route*) + *nbPointTouristiques*(*next*(*etapes*)))
}