Projet : “Conseil en restauration biologique et locale”

Dans ce projet, vous jouez le rôle d'une entreprise de conseil, qui met en relation ses clients, des restaurateurs avec des producteurs. Le but est de permettre aux restaurateurs de créer une carte à base de produits frais, biologiques et au meilleur bilan carbone possible.

Votre entreprise fournit à ses clients une API interrogeable par chacun d'entre eux. Cette API permet aux clients de s'identifier et de formuler leur requête (ingrédients, localisation de référence). Enfin, quand la requête est correctement formulée, le client peut demander à l'API de lui fournir une liste de producteurs.

Le code réalisé dans les TDs **??** à **??**, vous avance grandement dans ce travail. Il est à noter que des corrections de chacun de ces TDs est disponible sur moodle.

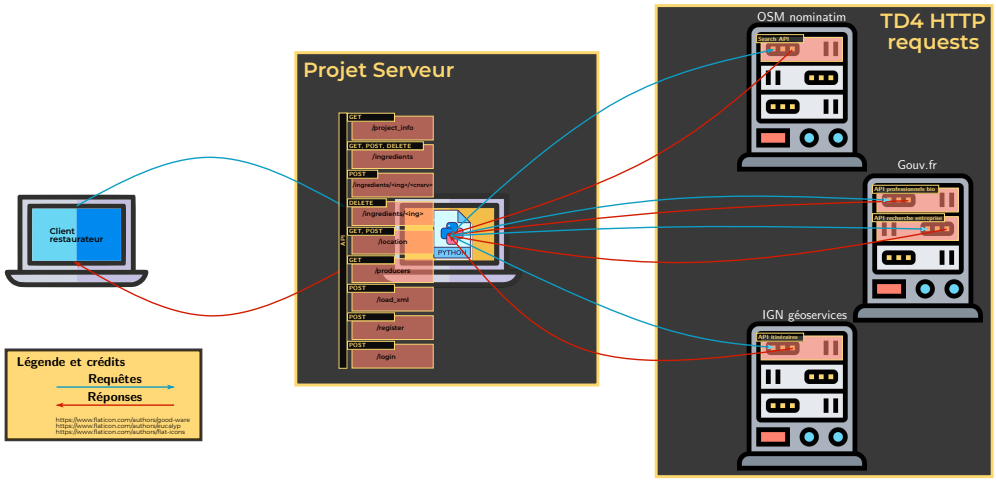

FIGURE 1 – Architecture Globale du projet

La partie sur laquelle vous êtes évalués est étiquetée « *Server Project* » dans la figure 1. Elle comprend le serveur et tous les traitements qu'il devra exécuter pour répondre aux requêtes des clients.

Cela signifie que le client de test que vous allez devoir créer pour tester votre serveur n'est pas à rendre. Nous n'évaluerons que le serveur.

1 Rendu

Le rendu (serveur) se fera via gitlab. Il faudra ajouter à votre projet les utilisateurs
- Audrey Serna
- Mathieu Loiseau
- Ludovic Moncla

Vous nous enverrez un e-mail pour nous donner l'url de votre projet.

2 Spécifications

Pour fournir les services demandés, votre serveur devra gérer un certain nombre de points de terminaison. Ces points de terminaisons appelleront des fonctions qui elles-mêmes utiliseront du code écrit dans les TDs précédents. Tous les points de terminaisons sont listés ci-dessous. À vous de voir si vous préférez les séparer — assigner une fonction à chaque couple `(route, méthode http)` — ou regrouper certains points de terminaisons qui traitent une même « route ».

Votre API devra suivre scrupuleusement les spécifications suivantes.

2.1 Port d'écoute

Votre serveur devra écouter le port `5080`.

2.2 Point de terminaison `/project_info GET`

Ce point de terminaison doit nous donner toutes les informations concernant votre projet :
- l'adresse du dépôt
- la liste des membres du projet
- quelle authentification vous avez mis en place (cf. section 3) : `null`, `"IP"` ou `"account"`
- quelle stratégie pour le stockage des données vous avez mis en place (cf. section 4) : `null`, `"serialisation"` ou `"sqlite"`

Description	Renvoie les informations concernant le projet	
Requête	Méthode	`GET`
	Paramètres	Ø
Réponse	Statut	200

	Format	JSON (`dict`) :
		`{"groupe":"GI3.1.4",`
		`"depot":"https://gitlab.insa-lyon.fr/cbd/g1-4",`
		`"authentification":"IP", /*ou "account" ou null*/`
		`"stockage":"serialisation", /* ou "sqlite" ou null*/`
		`"membres":[`
		`{"prenom":"John","nom":"Doe"},`
		`{"prenom":"Alain", "nom":"Connu"},`
		`/*...*/`
		`]`
		`}`

TABLE 1 – Point de terminaison `/project_info GET`

2.3 Route `/ingredients`

Cette route peut-être interrogée avec 3 méthodes différentes.

a) `GET`

Requête	Méthode	`GET`
	Paramètres	Ø
Réponse	Statut	200
	Description	Renvoie l'état actuel de la liste des ingrédients telle qu'elle a été saisie par le client.
	Format	JSON (`dict`) : `{"nom_ingrédient1":temps de conservation en jours, ...}`
	Exemple	`{"Pain frais":2, "Pommes de terre":30}`

TABLE 2 – Point de terminaison `/ingredients GET`

b) `POST`

Description	Remplace la liste des ingrédients par celle qui a été envoyée par le client et la renvoie	
Requête	Méthode	`POST`
	Paramètres	JSON (`dict`) : `{"nom_ingrédient1":tps conservation (j), ...}`
Réponse	Statut	200
	Format	JSON (`dict`) : `{"nom_ingrédient1":tps conservation (j), ...}`
	Exemple	`{"Pain frais":2, "Pommes de terre":30}`

TABLE 3 – Point de terminaison `/ingredients POST`

c) `DELETE`

Description	Efface la liste des ingrédients et renvoie un message textuel	
Requête	Méthode	`DELETE`
	Paramètres	Ø
Réponse	Statut	200
	Format	Text : Un message indiquant si la requête a marché

TABLE 4 – Point de terminaison `/ingredients DELETE`

2.4 Point de terminaison `/ingredients/<ing>/<cnsrv>` `POST`

Description	Ajoute un ingrédient à la liste et renvoie l'état actuel de la liste	
Requête	Méthode	`POST`
	Paramètres	dans l'url
Réponse	Statut	200 OK
		304 pas de modification
	Format	JSON (`dict`) : `{"nom_ingrédient1":tps conservation (j), ...}`
	Exemple	`{"Pain frais":2, "Pommes de terre":30}`

TABLE 5 – Point de terminaison `/ingredients/<ing>/<cnsrv>` `POST`

2.5 Point de terminaison `/ingredients/<ing>` `DELETE`

Description	Efface l'ingrédient spécifié de la liste des ingrédients et renvoie l'état de la liste	
Requête	Méthode	`DELETE`
	Paramètres	dans l'url
Réponse	Statut	200 OK
		304 pas de modification
	Format	JSON (`list`) : `["ingredient1","ingredient2", ...]`
	Exemple	`["Pain frais", "Pommes de terre"]`

TABLE 6 – Point de terminaison `/ingredients/<ing>` `DELETE`

2.6 Route `/location`

Cette route peut-être interrogée avec 2 méthodes différentes.

a) `GET`

Description	Renvoie l'état actuel de l'adresse de référence (L'adresse où l'on voudrait implanter un restaurant)	
Requête	Méthode	`GET`
	Paramètres	Ø
Réponse	Statut	200
	Format	JSON (`dict`) : `{"street":"address", "city":"name of the city"}`
	Exemple	`{"city":"Rennes", "street":"5, allée Geoffroy de Pontblanc"}`

TABLE 7 – Point de terminaison `/location` `GET`

b) POST

Description	Remplace l'adresse de référence par celle qui a été envoyée par le client et la renvoie	
Requête	Méthode	POST
	Paramètres	JSON (dict) : {"street":"address", "city":"name of the city"}
Réponse	Statut	200
	Format	JSON (dict) : {"street":"address", "city":"name of the city"}
	Exemple	{"city":"Rennes", "street":"5, allée Geoffroy de Pontblanc"}

TABLE 8 – Point de terminaison /location POST

2.7 Route /producers

Cette route peut-être interrogée avec 1 seule méthode.

a) GET

Description	Utilise les données d'entrées stockées sur le serveur (avec les routes ingredients et location) comme données d'entrées des fonctions créées dans le TD?? et renvoie au client un producteur pour chaque denrée, ainsi que la distance en km qui le sépare de l'adresse de référence		
Requête	Méthode	GET	
	Paramètres	Ø	
Réponse	Statut	200	OK
		400	des paramètres manquent
	Format	400	JSON (list) indiquant les paramètres manquants
		200	JSON (dict) associating to each product, a company, its manager's name and a distance in km
	Exemple	400	["location","ingredients"] ou ["ingredients"] ou ["location"]

		200	{
			"Pain frais":{
			"Entreprise":"MONTOIR MATHIEU",
			"Manager":"Mathieu Montoir",
			"Distance":20.6
			},
			"Pommes de terre":{
			"Entreprise":"ESPACE EMPLOI - JARDINS DU BREIL",
			"Manager":"Unknown",
			"Distance":4.57
			}
			}

TABLE 9 – Point de terminaison /producers GET

2.8 Point de terminaison `/load_xml POST`

Ce point de terminaison permet d'envoyer un fichier XML contenant toutes les données du problème (et ainsi de gérer la localisation et les ingrédients en une requête).

Pour mener à bien le traitement associé, il vous faudra adapter le parseur, et utiliser les exemples d'envoi de fichier disponibles sur moodle :

— file_upload_server.py
— file_upload_client.py

Description	Remplace l'adresse de référence et la liste des ingrédients d'après les données du fichier XML (structuré d'après le schéma du TD n°3) réceptionné.		
Requête	Méthode	`POST`	
	Paramètres	Envoi de fichier	
Réponse	Statut	200	OK
		400	Mauvaise requête (pas de fichier, fichier mal formé, etc.)
	Format	`text` : explication de l'erreur / `"OK"` en cas de succès	
	Exemple	`Response("Erreur lors du parsing du fichier", status=200)`	

TABLE 10 – Point de terminaison `/load_xml POST`

3 Authentification

Il faudra pouvoir gérer plusieurs utilisateurs, cela signifie que les données du problèmes seront associées à un utilisateur et un seul. Pour cela, on peut faire les choses de plusieurs manières (voir barème) :

1. utiliser l'adresse IP pour distinguer les utilisateurs (pb du changement d'IP, ou d'avoir plusieurs utilisateurs depuis la même adresse — ordinateur partagé, boxes) ;
2. on peut demander à l'utilisateur de se créer un compte, puis de s'authentifier.

La première solution ne demande pas de point de terminaison spécifique, c'est celle que nous attendons par défaut.

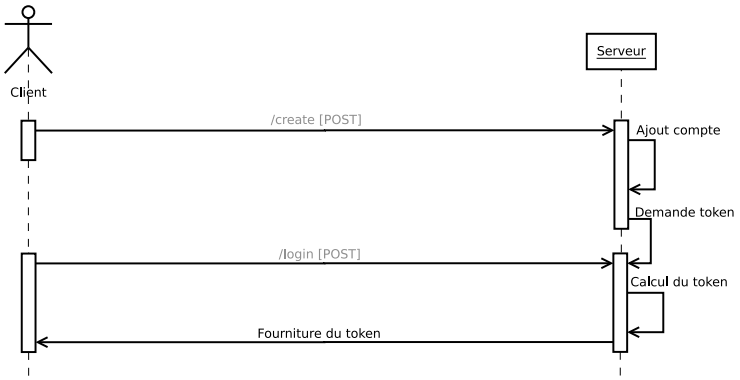

FIGURE 2 – Diagramme de séquence de l'authentification avec un compte

La seconde est plus complexe, mais plus proche du réel (quoi qu'on utiliserait des fonctionnalités d'authentifications intégrées à flask [1]). Pour traiter cette stratégie, on suivra le diagramme de séquence de la figure 2, cela rajoute deux points de terminaison et demande d'ajuster tous les autres.

3.1 Route `/register POST`

Description	L'utilisateur se choisit un identifiant de compte. S'il existe déjà le serveur renvoie une erreur, sinon il lui renvoie un token d'authentification.		
Requête	Méthode	POST	
	Paramètres	JSON (dict) : `{"login":"username", "password":"1dGZ*neo8P!"}`	
Réponse	Statut	200	La création de compte a réussi
		400	Échec de la création de compte (pseudo existant ou non fourni)
	Format	200	text : token d'identification aléatoire au format `(\d\w[%*:.-~=]){10}`
		400	JSON (dict) : `{"error":"explication de l'erreur"}`
	Exemple	200	text : `"1Rv*3Rt4c%"`
		400	JSON (dict) : `{"error":"user name already exists"}`

TABLE 11 – Point de terminaison `/register POST`

3.2 Route `/login POST`

Description	L'utilisateur se connecte en utilisant son login et son mot de passe. Si l'authentification est réussie, le serveur renvoie un nouveau token. Sinon il envoie une erreur.		
Requête	Méthode	POST	
	Paramètres	JSON (dict) : `{"login":"username", "password":"1dGZ*neo8P!"}`	
Réponse	Statut	200	Le login a réussi
		401	Échec du login
	Format	200	text : token d'identification aléatoire au format `(\d\w[%*:.-~=]){10}`
		400	JSON (dict) : `{"error":"explication de l'erreur"}`
	Exemple	200	text : `"13Rt4c%Rv*"`
		401	JSON (dict) : `{"error":"bad login/password combination"}`

TABLE 12 – Point de terminaison `/login POST`

3.3 Conséquences

Attention Pour les groupes qui choisiront de déployer l'authentification, cette méthode aura des conséquences sur **tous** les autres points de terminaison. Il devront tous vérifier que des données JSON sont envoyées, comprennent un champ `"token"` et que ce dernier contient une valeur associée à un compte utilisateur existant. Si le token n'est pas bon, il faudra renvoyer une erreur `401`.

1. https://flask-login.readthedocs.io/en/latest/

4 Stockage des données sur le serveur

Afin que les utilisateurs ne reprennent pas de 0 à chaque redémarrage du serveur. Il serait pertinent qu'il conserve des données. Cela peut être fait aussi bien en utilisant la sérialisation (cf. TD 2) qu'une base de données (ex : SQLite, cf. section 5 du TD 5).

5 Barême

Le barême est indicatif : le nombre de points associé à un point de terminaison peut fluctuer. Les nombres de points indiqués partent du principe que les points de terminaison suivent les spécifications et fonctionnent. Il est possible qu'un point de terminaison fonctionnel qui ne suit pas les spécification soit évalué à 0.

Le barême propose un ordre de résolution du problème.

Note	**Fonctionnalité suivant les spécifications**
1,5	`/project_info GET`

4	`/ingredients GET`
	`/ingredients/<ing>/<csrv> POST`
	`/location GET`

2	`/location POST`
4+2	`/producers GET` (2 points hors-barême pour les optimisations)
1,5	`/ingredients DELETE` `/ingredients POST`
0,5	`/ingredients/<ingred> DELETE`
4	`/load_xml POST`
2	stratégie de stockage des données

0,5+2	stratégie d'authentification (2 points hors-barême pour l'authentification)
	`/register POST`
	`/login GET`

TABLE 13 – Barême indicatif **(la note maximale est 20)**