1. Choix technologique : Flask (Python)

Le framework Flask a été choisi pour plusieurs raisons :

Simplicité et légèreté

Flask est un framework minimaliste, ce qui permet :

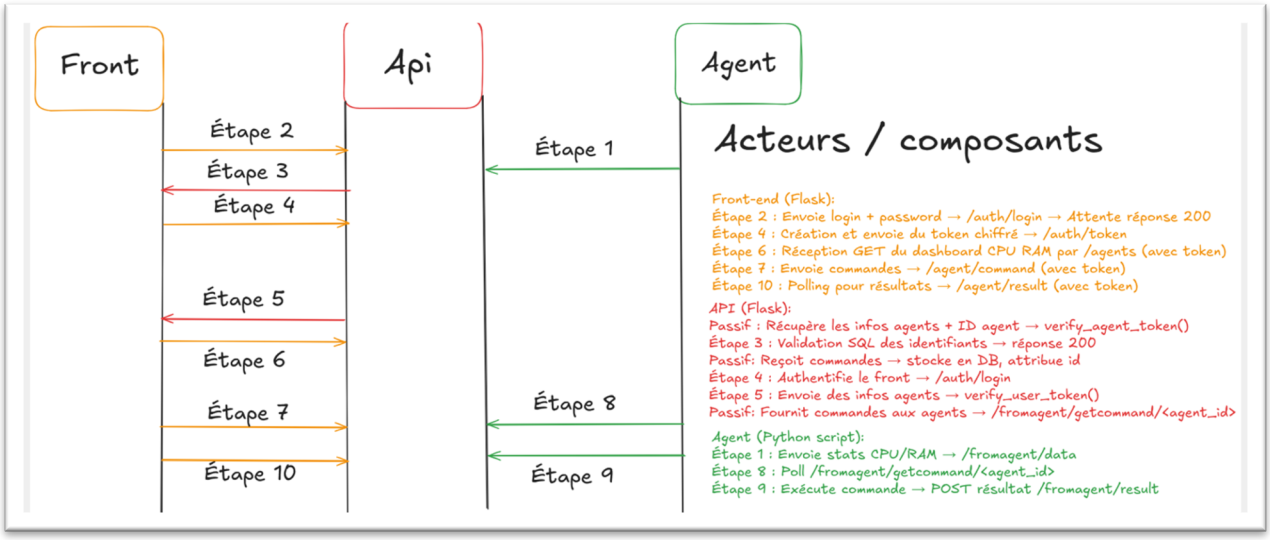
- une compréhension rapide de l'architecture,
- une implémentation rapide,
- une bonne lisibilité du code.

Compatibilité avec Python

L'agent client étant également développé en Python, cela permet :

- une homogénéité technologique,
- une meilleure maintenabilité,
- une facilité de développement.

2. Architecture de communication



3. Gestion de la base de données

La base contient plusieurs tables :

Table agents

- id
- cpu
- ram

- last_seen
- status

Table commands

- id
- agent_id
- command
- result
- status

Table users

- username
- password (hashé)
- token

4. Rôle de l'API dans l'architecture globale

L'API constitue le composant central du système. Elle agit comme un intermédiaire sécurisé entre :

- les agents installés sur les machines contrôlées,
- l'interface web utilisée par l'utilisateur,
- la base de données qui stocke les informations.

Elle est implémentée en Python à l'aide du framework Flask, qui permet de créer facilement des endpoints HTTP et de structurer une architecture sécurisée.

Elle remplit plusieurs fonctions essentielles :

➔ Réception des informations envoyées par les agents

La réception des informations système envoyées par les agents est gérée par la route :

```
@app.route("/fromagent/data", methods=["POST"])
def update_agent_status():
```

Cette fonction est appelée périodiquement par l'agent client. Elle reçoit des informations au format JSON contenant notamment :

- l'identifiant de l'agent (`agent_id`)
- l'utilisation CPU (`cpu`)
- l'utilisation mémoire (`ram`)

Avant de traiter les données, l'API vérifie que l'agent est autorisé à communiquer grâce à la fonction :

```
verify_agent_token():
```

Cette fonction récupère le token présent dans l'en-tête HTTP Authorization et vérifie qu'il correspond au token attendu. Si le token est invalide ou absent, la requête est rejetée avec un code 401 Unauthorized. Ce mécanisme permet d'empêcher des machines non autorisées d'envoyer des données à l'API.

Une fois le token vérifié, la fonction met à jour les informations de l'agent dans la base de données. Si l'agent existe déjà, ses informations sont mises à jour. Sinon, un nouvel agent est enregistré. Un horodatage est également ajouté afin de suivre la dernière activité de l'agent.

➔ Transmission des commandes aux agents

La transmission des commandes est réalisée via la route :

```
@app.route("/agent/command", methods=["POST"])
def add_command():
```

Cette fonction est appelée par le frontend lorsqu'un utilisateur souhaite exécuter une commande sur un agent. La commande est reçue au format JSON et enregistrée dans la base de données avec l'identifiant de l'agent cible.

La commande est stockée avec un statut initial « pending », ce qui signifie qu'elle est en attente d'exécution.

➔ Récupération des commandes par l'agent

Les agents interrogent régulièrement l'API afin de vérifier si une commande leur est destinée, via la route :

```
@app.route("/fromagent/getcommand/<agent_id>", methods=["GET"])
def get_command(agent_id):
```

Cette fonction vérifie d'abord l'authentification grâce à la fonction :

```
verify_agent_token():
```

Elle exécute ensuite une requête SQL pour rechercher une commande. Si une commande est trouvée, elle est renvoyée à l'agent au format JSON. Sinon, aucune commande n'est retournée.

Ce mécanisme repose sur un modèle où l'agent interroge périodiquement le serveur pour obtenir de nouvelles instructions.

➔ Réception des résultats d'exécution

Une fois la commande exécutée localement, l'agent envoie le résultat à l'API via la route :

```
@app.route("/fromagent/result", methods=["POST"])
def submit_command_result():
```

Cette fonction vérifie également le token d'authentification

Elle récupère ensuite le résultat envoyé par l'agent et met à jour la base de données. Le statut de la commande est changé en « done », ce qui indique qu'elle a été exécutée.

Ce mécanisme permet de conserver un historique des commandes et de leurs résultats.

➔ Authentification des utilisateurs

L'authentification des utilisateurs est gérée par la route :

```
@app.route("/auth/login", methods=["POST"])
def login():
```

Cette fonction reçoit le nom d'utilisateur et le mot de passe, puis applique une fonction de hachage SHA-256. Ce mécanisme permet de ne jamais stocker ni comparer les mots de passe en clair, ce qui améliore la sécurité du système.
Si les identifiants sont valides, un token est retourné à l'utilisateur. Ce token est ensuite utilisé pour authentifier les requêtes suivantes.

➔ Centralisation et gestion des données

Toutes les opérations de lecture et d'écriture dans la base de données sont réalisées via la classe Database
Cette abstraction permet de centraliser les accès à la base de données et de simplifier la gestion des agents, des utilisateurs et des commandes.
La base de données permet notamment de stocker :

- les agents connectés
- les commandes envoyées
- les résultats des commandes
- les utilisateurs

➔ Sécurisation des communications

Plusieurs mécanismes ont été mis en place pour sécuriser l'API.
Tout d'abord, un système de token est utilisé pour authentifier les agents, via la fonction `verify_agent_token()`. Cela permet d'éviter qu'un agent non autorisé puisse communiquer avec le serveur.
Ensuite, les mots de passe utilisateurs sont stockés sous forme de hash SHA-256, ce qui empêche leur récupération en clair en cas de compromission de la base de données.

L'API constitue le cœur du système et assure la communication entre les agents et l'interface web. Elle permet de recevoir les informations système, de transmettre les commandes, de récupérer les résultats et de gérer l'authentification.

Cette architecture permet de mettre en place un système centralisé, structuré et sécurisé, adapté à un environnement pédagogique et conforme aux principes d'une architecture client-serveur.

5. Rôle de l'agent client dans l'architecture globale

L'agent client constitue le composant exécuté sur la machine supervisée. Il agit comme un intermédiaire local sécurisé entre :

- la machine contrôlée, dont il collecte les informations système,
- l'API centrale, qui coordonne les commandes et stocke les données,
- l'utilisateur via l'interface web, indirectement par l'API.

Il remplit plusieurs fonctions essentielles :

- collecte des informations système (CPU, RAM, etc.) et envoi à l'API via l'endpoint `/fromagent/data`,
- réception des commandes depuis l'API via `/fromagent/getcommand/<agent_id>`,
- exécution locale sécurisée des commandes via la fonction `simple_command()`,
- transmission des résultats d'exécution à l'API via `/fromagent/result`,
- maintien d'une communication continue avec l'API et gestion des erreurs en cas d'indisponibilité du serveur,
- respect du principe du moindre privilège : aucune élévation de droits, aucune persistance et exécution uniquement avec les droits de l'utilisateur courant.

L'agent est développé en Python, garantissant **une cohérence technologique avec l'API** et facilitant la maintenance et la compréhension globale du système. Il repose sur un modèle dans lequel il interroge périodiquement l'API pour envoyer ses statistiques et récupérer les commandes, ce qui assure une communication simple et fiable.

Choix technologique : Python pour l'agent client

Le choix de Python pour l'agent client s'explique par plusieurs raisons :

Compatibilité et homogénéité
L'API étant développée en Python avec Flask, utiliser Python pour l'agent client permet :

- une cohérence technologique sur l'ensemble du projet,
- une meilleure lisibilité et compréhension du code pour tous les membres du groupe,
- une maintenance facilitée et une évolution plus simple du système.

Simplicité et rapidité de développement
Python offre des bibliothèques standard et spécialisées pour gérer les besoins du projet :

- `psutil` pour collecter facilement les informations système (CPU, RAM, processus),
- `subprocess` pour exécuter les commandes locales dans un environnement contrôlé,
- `requests` pour communiquer avec l'API via HTTP de manière sécurisée.

Ces bibliothèques permettent de développer rapidement un agent fonctionnel tout en garantissant la sécurité et la traçabilité des actions.

Flexibilité et pédagogie
Python est simple à apprendre et à lire, ce qui est un avantage pédagogique. Cela permet au groupe de comprendre et de justifier facilement le fonctionnement de l'agent, notamment la collecte des données, l'exécution des commandes et l'envoi des résultats à l'API.

6. Interface Web (FrontEnd)

L'interface web constitue le point d'interaction entre l'utilisateur et le système d'agent pédagogique. Elle permet à l'utilisateur authentifié de visualiser les agents connectés, de consulter leurs informations système et d'envoyer des commandes à distance. Le frontend a été développé en Python à l'aide du framework Flask, qui permet de gérer les routes, les sessions et le rendu dynamique des pages HTML.

Le frontend ne communique jamais directement avec les agents. Toutes les interactions passent par l'API centrale, ce qui garantit une architecture sécurisée et une séparation claire des responsabilités. Cette approche permet de limiter la surface d'attaque et d'éviter toute communication directe non contrôlée avec les machines surveillées.

L'authentification des utilisateurs est réalisée via la route login(), qui envoie les identifiants à l'API pour vérification. Si l'authentification est valide, un token est retourné et stocké dans la session utilisateur (session["token"]). Ce token est ensuite inclus dans les en-têtes des requêtes envoyées à l'API, ce qui permet de garantir que seules les requêtes authentifiées peuvent accéder aux ressources sensibles.

La route dashboard() permet de récupérer la liste des agents connectés en envoyant une requête GET à l'API. Les informations reçues, telles que l'identifiant de l'agent et l'utilisation des ressources, sont ensuite affichées dynamiquement dans l'interface. Ce mécanisme permet à l'utilisateur de superviser l'état des agents en temps réel.

Le frontend intègre également un terminal interactif implémenté via la fonction terminal(). Lorsque l'utilisateur saisit une commande, celle-ci est transmise à l'API, qui la stocke et la transmet à l'agent concerné. Le frontend interroge ensuite régulièrement l'API afin de récupérer le résultat de l'exécution. Ce mécanisme repose sur un modèle de polling, permettant de simuler un terminal distant accessible depuis un navigateur.

Cette architecture garantit que le frontend agit uniquement comme une interface de présentation, sans accès direct aux ressources critiques. La centralisation des communications

via l'API permet de renforcer la sécurité, d'assurer le contrôle des accès et de maintenir une architecture modulaire et évolutive.

7. Choix technologique : Flask et non FastAPI

Initialement, le projet prévoyait l'utilisation de FastAPI pour le développement de l'API. Cependant, au cours du développement du frontend, qui a été réalisé en Flask, nous avons décidé de conserver Flask pour toute l'architecture backend afin de garantir une cohérence technologique. Ce choix répond à plusieurs objectifs pédagogiques et pratiques :

Apprentissage et compétences
Flask est un framework plus simple et minimaliste que FastAPI. Le choix de Flask nous a permis :

- de mieux comprendre la structuration d'une application web et la création d'API REST,
- de développer nos compétences sur un outil largement utilisé en milieu professionnel et pédagogique,
- de maintenir la lisibilité et la clarté du code, ce qui est essentiel dans un projet pédagogique.

Homogénéité technologique
L'API et l'agent client sont tous deux développés en Python. Cette cohérence permet :

- une meilleure maintenabilité du projet,
- une compréhension rapide des interactions entre l'API et l'agent,
- une simplification du déploiement et de la gestion des dépendances.

Flexibilité et pédagogie
Flask offre une grande flexibilité pour gérer les routes, l'authentification, les sessions et la communication avec le frontend. Son utilisation a permis de créer un système sécurisé et modulable, tout en restant accessible pour l'apprentissage et l'expérimentation.