



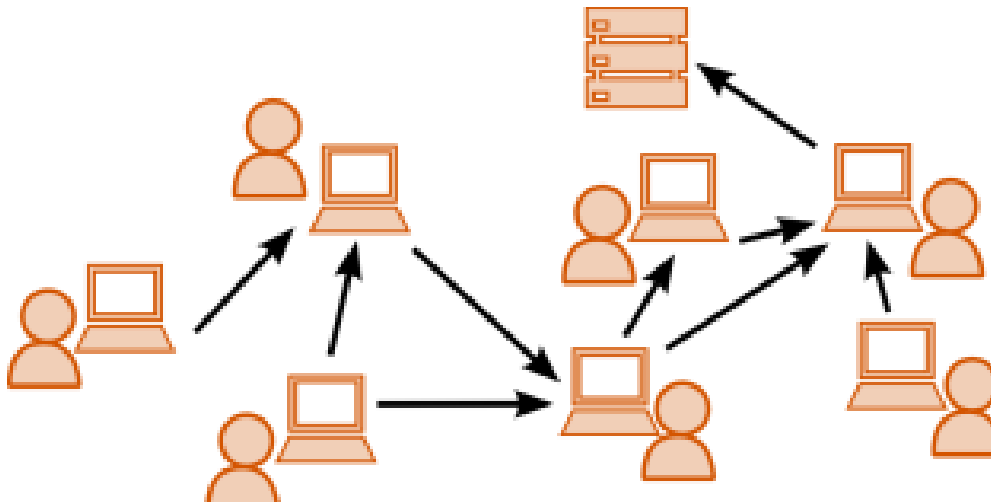
**ECOLE SUPERIEURE MULTINATIONALE  
DES TELECOMMUNICATIONS**

**Cycle : INGENIEUR DE CONCEPTION DES TELECOMS**

**Spécialité : INGENIERIE DES DONNEES ET INTELLIGENCE ARTIFICIELLE**

## **Projet du cours de Bases de systèmes distribués**

**Thème : Développer une exécution parallèle en utilisant les services web REST**



### **Réalisé par :**

Francklin Powell NIKIEMA

Joshua Juste NIKIEMA

Mafoya Elie ADJOBO

Sofiane ISSOUFOU ISSAKA

### **Proposé par :**

Dr. Modou GUEYE

## Table des matières

Introduction.....	3
I. Description de l'outil utilisé : FastAPI .....	4
1. Définition d'API REST.....	4
2. Pourquoi FastAPI ?.....	5
II. Description des URLs des services REST.....	6
1. Architecture de notre système distribué.....	6
2. Description des Endpoints et méthodes de chaque entité de notre système .....	6
Conclusion .....	22

# Introduction

Le domaine en constante évolution des systèmes distribués a considérablement redéfini la manière dont les tâches informatiques sont abordées au sein des réseaux interconnectés. Ce projet, réalisé dans le cadre du cours sur les Bases des Systèmes Distribués, vise à approfondir notre compréhension de ces systèmes en mettant en œuvre une exécution répartie au moyen de services web REST. En transposant les concepts explorés dans le cours, nous envisageons de démontrer les avantages et les nouvelles opportunités offertes par cette approche moderne et adaptable. Ce rapport détaille notre démarche pour mettre en place un système de traitement réparti à l'aide de services web REST, ouvrant la voie à des perspectives passionnantes dans le domaine des systèmes distribués.

# I. Description de l'outil utilisé : FastAPI

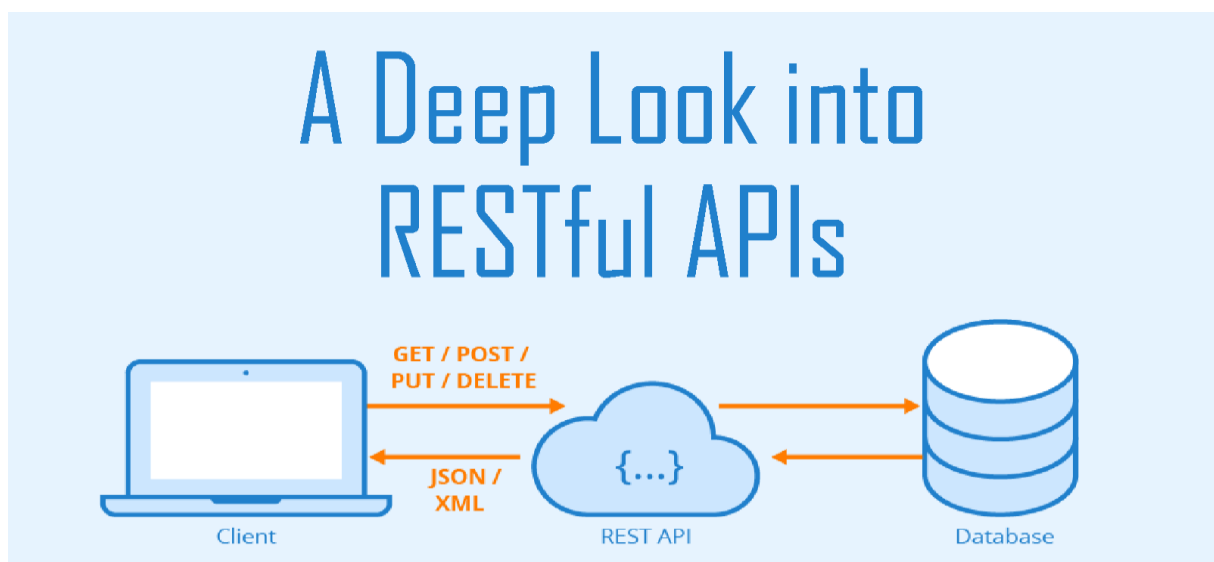
Dans l'univers du développement logiciel, foisonnent une multitude de cadres conceptuels s'appuyant sur une pléthore de langages de programmation. Parmi ces joyaux technologiques, nous trouvons d'éminents noms tels que le vénérable Spring de Java, l'élégant Express.js propulsé par Node.js, l'incontournable Ruby on Rails pour les passionnés de Ruby, sans oublier les remarquables Flask, FastAPI et Django du royaume Python.

C'est au sein de ce panorama éblouissant que notre chronique prendra place, plongeant avec audace dans l'univers de Python et embrassant la prestigieuse Framework FastAPI comme précieux allié dans cette quête de l'excellence.

## 1. Définition d'API REST

Une API, qui signifie "Application Programming Interface", représente un ensemble de fonctions informatiques qui permettent à deux logiciels de communiquer directement, sans intervention humaine. Une API peut être soit publique, accessible à tous, soit privée, réservée à certains utilisateurs. Son rôle est de fournir des services et de mettre à disposition des fonctionnalités ou des données.

En parallèle, les développeurs créent des programmes qui utilisent ces APIs pour interagir avec les services qu'elles exposent. Pour ce faire, ils se réfèrent à la documentation des APIs, qui leur fournit toutes les informations nécessaires pour utiliser efficacement ces interfaces de programmation.



Une API REST, abréviation de "Representational State Transfer Application Program Interface", incarne un style architectural qui facilite la communication entre logiciels, que ce soit sur un réseau ou au sein d'un même appareil. Principalement destinées à la création de services web, ces APIs sont souvent qualifiées de "RESTful" et font

usage des méthodes HTTP pour échanger des données entre un client et un serveur, même si ces derniers utilisent des systèmes d'exploitation et des architectures différentes.

Le client peut solliciter des ressources en utilisant un langage compréhensible par le serveur, qui renvoie ensuite la ressource dans un format accepté par le client, couramment JSON ou XML. Pour identifier ces ressources, REST se repose sur les URI (Uniform Resource Identifier).

Ces ressources représentent des informations variées telles que des images, des documents, des personnes, et bien d'autres encore. L'approche REST offre ainsi un mécanisme flexible et uniforme pour interagir avec les services web, facilitant l'échange d'informations entre les parties prenantes.

## 2. Pourquoi FastAPI ?

FastAPI : L'étoile montante des Frameworks Python pour le développement d'APIs

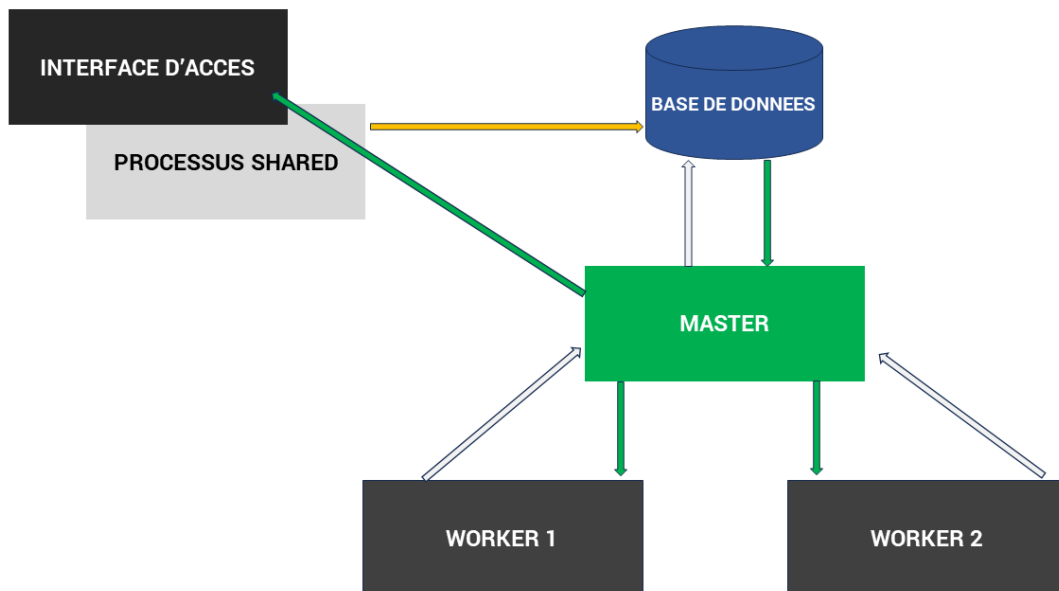
Lorsqu'il s'agit de choisir le cadre conceptuel idéal pour la création d'applications web et d'interfaces de programmation (APIs RestFul), FastAPI émerge comme une étoile montante incontestable dans l'univers foisonnant des Frameworks Python. Cette pépite technologique a su conquérir les cœurs des développeurs grâce à une panoplie de caractéristiques remarquables qui en font un choix judicieux et éclairé.

Comme avantages :

- Très facile à utiliser
- Haute performance et rapidité.
- Documentation automatique et interactive.
- Intégration fluide avec l'écosystème Python.
- Sécurité renforcée grâce à la validation de types.
- Communauté active et en constante évolution.

## II. Description des URLs des services REST

### 1. Architecture de notre système distribué



### 2. Description des Endpoints et méthodes de chaque entité de notre système

Notre application est constituée d'un master qui est responsable de la coordination, de la gestion et de la distributions des tâches aux esclaves. Nous avons par la suite simulé deux workers qui exécutent les tâches provenant du master. Notre description des Endpoints et des méthodes de chaque entité de notre système se fera en deux parties : une partie concernant le master et une autre basée sur les workers.

#### ➤ Descriptions des Endpoints et des méthodes du master

- Explication du code model.py

Nous avons dans cette partie un script nommé **model.py**. Ce programme commence par l'importation des modules pydantic, sqlite3, et socket.

```
Master > models > model.py > Fruit
1  from pydantic import BaseModel
2  import sqlite3
3  import socket
4
5
```

- **pydantic** : C'est un module utilisé pour la validation de données et la définition de modèles de données. Il permet de définir des structures de données avec des contraintes et des types. Il va dans notre cas importer les éléments des classes Fruit et INGREDIENT.
- **sqlite3** : C'est un module pour interagir avec des bases de données SQLite. SQLite est un moteur de base de données léger et largement utilisé, souvent intégré dans des applications pour stocker des données.
- **socket** est généralement utilisé pour la communication réseau, comme la création de sockets TCP/UDP.

Nous allons ensuite définir une fonction **ip\_adress()** qui utilise le module socket pour obtenir l'adresse IP associée au nom d'hôte de la machine.

```

def ip_adress():
    # On récupère le nom d'hôte de la machine
    hostname = socket.gethostname()

    # On récupère l'adresse IP associée au nom d'hôte
    ip_adress = socket.gethostbyname(hostname)

    return ip_adress

```

Après cette étape, nous allons définir deux classes **Fruit** et **INGREDIENT** en utilisant le pydantic affecté à la sous classe **BaseModel**.

```

class Fruit(BaseModel):
    fruit: str

class INGREDIENT(BaseModel):
    id: int
    fruit: str
    execution_time: int

db_path = "commands.db"

```

#### ❖ Définition de la classe Fruit

La classe **Fruit** est défini en tant que sous-classe de **BaseModel**, ce qui indique qu'elle hérite des fonctionnalités de validation et de gestion des données fournies par **BaseModel**.

La classe Fruit a un attribut fruit de type **str**, ce qui signifie qu'elle représente un objet de fruit avec une propriété fruit contenant une chaîne de caractères.

#### ❖ Définition de la classe INGREDIENT

La classe **INGREDIENT** est également définie en tant que sous-classe de **BaseModel**.

La classe **INGREDIENT** a trois attributs :

- ✓ **id**: de type **int**, représentant un identifiant numérique.
- ✓ **fruit**: de type **str**, indiquant le nom du fruit lié à cet ingrédient.
- ✓ **execution\_time**: de type **int**, représentant le temps d'exécution associé à cet ingrédient.

- ❖ **Définition du chemin de la base de données** : La variable **db\_path** est définie avec le chemin relatif "**commands.db**". Cela suggère qu'elle pourrait être utilisée pour stocker ou accéder à une base de données SQLite appelée "**commands.db**".

Nous allons par la suite définir une fonction **create\_table** destinée à créer une table dans une base de données SQLite en utilisant le chemin de la base de données spécifié par la variable **db\_path**.

```
def create_table():
    connection = sqlite3.connect(db_path)
    cursor = connection.cursor()
    cursor.execute("DROP TABLE IF EXISTS commands")
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS commands (
            message TEXT NOT NULL,
            time INTEGER NOT NULL
        )
    """)
    connection.commit()
    connection.close()
```

- ❖ **Ouverture de la connexion à la base de données**

La première ligne de la fonction crée une connexion à la base de données **SQLite** en utilisant le chemin **db\_path** précédemment défini.

- ❖ **Création du curseur**

Un **curseur** est créé pour interagir avec la base de données. Le **curseur** est utilisé pour exécuter des requêtes SQL et récupérer les résultats.

- ❖ **Suppression de la table si elle existe**

Avant de créer la table, le code exécute une **requête SQL** pour supprimer la table "**commands**" si elle existe déjà. Cela permet de s'assurer que la table est recréée à partir de zéro, évitant ainsi les conflits potentiels.

- ❖ **Création de la table "commands"**

Une fois la table supprimée (si elle existait), le code exécute une requête SQL pour créer la table "**commands**". Cette table a deux colonnes :



- ✓ **message** : de type TEXT (chaîne de caractères) et non nulle, pour stocker un message.
- ✓ **time** : de type INTEGER (entier) et non nul, pour stocker un horodatage.
- ❖ Validation des changements et fermeture de la connexion

Après avoir exécuté les requêtes SQL pour créer et configurer la table, la connexion est validée (commit) pour enregistrer les modifications dans la base de données. Ensuite, la connexion à la base de données est fermée.

De plus nous avons implémenter la fonction **insert\_command** destiné à insérer des données dans la table "commands" d'une base de données SQLite en utilisant les paramètres **message**, **time** et **db\_path**.

```
def insert_command(message, time, db_path):
    connection = sqlite3.connect(db_path)
    cursor = connection.cursor()
    cursor.execute("""
        INSERT INTO commands (message, time) VALUES (?, ?)
    """, (message, time))
    connection.commit()
    connection.close()
```

Pour terminer, nous avons implémenter la fonction **get\_commands** destiné à récupérer les données de la table "commands" dans une base de données SQLite en utilisant le chemin **db\_path** fourni en tant que paramètre.

```
def get_commands(db_path):
    connection = sqlite3.connect(db_path)
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM commands")
    commands = cursor.fetchall()
    cursor.execute("DELETE FROM commands")
    connection.commit()
    connection.close()

    return commands
```

### • Explication du code master api.py

Ce programme retrace toutes les activités principales de notre API interconnectée à notre base de données **commands**.

On commence tout d'abord avec l'importation des modules et dépendances liés à **FastAPI**, à la définition de modèles et à l'envoi de requêtes **HTTP**.

```

from fastapi import APIRouter
from fastapi.responses import JSONResponse
import time
from typing import Optional
from models.model import *
import random
import requests

```

- ✓ **APIRouter** : C'est une classe de FastAPI qui permet de définir des routes pour un routeur spécifique.
- ✓ **JSONResponse** : C'est une classe de FastAPI qui permet de retourner des réponses JSON.
- ✓ **time** : Ce module fournit des fonctions pour travailler avec le temps.
- ✓ **Optional** : C'est un type d'annotation qui indique qu'un argument ou un attribut peut être optionnel.
- ✓ **models.model** : Le chemin relatif models.model indique que le module est situé dans un sous-répertoire models sous le répertoire courant.
- ✓ **random** : Ce module permet de générer des nombres aléatoires.
- ✓ **requests** : C'est une bibliothèque Python qui facilite l'envoi de requêtes HTTP.

Nous allons par la suite créer un routeur **FastAPI (router)** et définir une adresse partagée (**shared\_address**) utilisée pour obtenir la liste des tâches à faire à partir d'un autre service ou d'une ressource.

```

# Create a FastAPI router
router = APIRouter()

# Shared address to get the list of tasks to do
shared_address = "http://192.168.131.251:8001/get_tasks_to_do/"

```

### ❖ Création d'un routeur FastAPI

La première ligne crée un objet router en utilisant la classe APIRouter fournie par FastAPI. Les routeurs FastAPI sont utilisés pour organiser les routes et les fonctionnalités spécifiques de l'API.

### ❖ Définition de l'adresse partagée (shared\_address)

La deuxième ligne définit une chaîne de caractères shared\_address contenant l'URL [http://192.168.131.251:8001/get\\_tasks\\_to\\_do/](http://192.168.131.251:8001/get_tasks_to_do/). Cette adresse est utilisée pour obtenir la liste des tâches (des fruits) à faire à partir d'un autre service.

Par la suite, plusieurs variables sont déclarées pour gérer les tâches et leur état.

```
# Variables to manage tasks and their state
start_time = None
reponse_shared = None
Fruits = None
tasks_to_do = []
tasks_being_done = {}
command_dict = {}
task_already_done = []
task_len = None
```

Après cette étape, nous allons définir un **événement** qui se déclenche lors du démarrage de l'application **FastAPI**, et cet événement exécute la fonction **create\_table()** pour créer la table dans la base de données si elle n'existe pas déjà.

```
# Event on startup
@router.on_event("startup")
async def startup_event():
    """
    Event that is triggered on application startup.
    """
    create_table() # Create the table if it doesn't exist in the database
```

Par la suite, nous allons définir une route **POST** dans le routeur **FastAPI (router)** pour envoyer une tâche à un travailleur (**worker**) en utilisant l'adresse partagée (**shared\_address**).

```
# Send a task to a worker
@router.post("/send_task")
async def send_task(nom_worker: dict):
    """
    Send a task to a worker.

    Parameters:
    - nom_worker (dict): A dictionary containing the worker's name.

    Returns:
    - JSONResponse: JSON response containing the task details if available, or a message if no tasks
    """
    global tasks_to_do, start_time, tasks_being_done, reponse_shared, task_len, task_already_done
    nom_worker = nom_worker.get("nom_worker")
    reponse_shared = requests.get(shared_address)
```

Nous poursuivons le code par la vérification des conditions :

- ✓ Si la réponse obtenue depuis l'adresse partagée (**reponse\_shared**) a un code d'état **HTTP** de 200 (ce qui signifie que la requête a réussi). S'il en est ainsi, il utilise les données **JSON** reçues pour initialiser certaines variables liées aux tâches et aux ingrédients.

```

if reponse_shared.status_code == 200:
    Fruits = reponse_shared.json()
    task_len = len(Fruits)
    tasks_to_do = [INGREDIENT(id=i, fruit=ingredient['fruit'], execution_time=random.randint(4, 20))
                    for i in range(task_len)]
    start_time = None
    task_already_done = []
    tasks_being_done = {}

```

Nous allons par la suite gérer la logique d'attribution des tâches aux travailleurs en fonction de l'état des tâches disponibles.

```

if start_time is None:
    start_time = time.time()

if tasks_to_do:
    task = tasks_to_do.pop(0)
    tasks_being_done[task.id] = task
    print(f"Sending task [id: {task.id}, fruit: {task.fruit}] to worker {nom_worker}")
    return JSONResponse(content={"id": task.id, "fruit": task.fruit, "execution_time": task.execution_time})
else:
    print(f"The worker {nom_worker} asks for more tasks to do")
    return JSONResponse(content={"message": "No more tasks"}, status_code=202)

```

Nous poursuivons notre code en gérant la réception des résultats d'une tâche depuis un travailleur. Il met à jour les données relatives aux tâches terminées et en cours d'exécution.

```

# Receive a result from a worker
@router.post("/result")
async def receive_result(result_data: Optional[dict]):
    """
    Receive a result from a worker.

    Parameters:
    - result_data (Optional[dict]): A dictionary containing the result data including the task and result

    Returns:
    None
    """
    global tasks_to_do, start_time, tasks_being_done, task_already_done, task_len

    task = dict(result_data.get("task"))
    result = result_data.get("result")

    task_already_done.append(task.get("id"))
    tasks_being_done.pop(int(task.get("id")), None)

```

Après cette étape nous allons traiter les résultats des tâches reçus des travailleurs et effectuer certaines actions en fonction de l'état des tâches à faire et en cours d'exécution.

```

if not tasks_to_do and not tasks_being_done:
    print("Task to do: ", tasks_to_do, "\nTask being done : ", tasks_being_done)
    end_time = time.time()
    command_dict = {
        "message": "The salad is ready! Enjoy!",
        "preparation_time": end_time - start_time
    }
    insert_command(command_dict["message"], command_dict["preparation_time"], db_path)
else:
    print(f"Task: {result}; came from worker {task.get('nom_worker')}")

```

Pour terminer, nous allons définir une route **GET** dans le routeur **FastAPI (router)** pour obtenir une commande à envoyer. Cette route est utilisée pour obtenir une commande à partir de la base de données, et si une commande est disponible, elle renvoie les détails de la commande sous forme de réponse **JSON**.

```

# Get a command to send
@router.get("/send_command")
async def send_command():
    """
    Get a command to send.

    Returns:
    - JSONResponse: JSON response containing the command details if available, or a message if no com
    """
    command_list = get_commands(db_path)

    if len(command_list) != 0:
        command_dict = {
            "message": command_list[0][0],
            "preparation_time": command_list[0][1]
        }
        return JSONResponse(content=command_dict, status_code=200)
    else:
        return JSONResponse(content={"message": "No command"}, status_code=202)

```

### ✓ Le code main.py

Ce code définit et exécute une application FastAPI en utilisant le routeur et les routes que vous avez définis précédemment.

```

main.py > ...
1  from fastapi import FastAPI
2  from routes.master_api import router
3  import uvicorn
4  #from models.model import ip_address
5
6  app = FastAPI()
7
8  app.include_router(router)
9
10 if __name__ == "__main__":
11     uvicorn.run(app, host="192.168.131.63", port=8000)

```

### ❖ Importation des modules et des dépendances

Le code importe les modules et les dépendances nécessaires, notamment la classe FastAPI, le routeur router que vous avez défini dans le fichier `routes.master_api`, la bibliothèque uvicorn pour le serveur ASGI et la fonction `ip_address` que vous avez définie dans le fichier `models.model`.

#### ❖ Création d'une instance de l'application FastAPI (app)

La ligne `app = FastAPI()` crée une instance de la classe FastAPI, qui sera utilisée pour définir les routes et les fonctionnalités de l'application.

#### ❖ Inclusion du routeur dans l'application

La ligne `app.include_router(router)` inclut le routeur router que vous avez défini précédemment dans l'application. Cela signifie que les routes définies dans le routeur seront accessibles à partir de cette application.

#### ❖ Exécution de l'application (if `__name__ == "__main__":`)

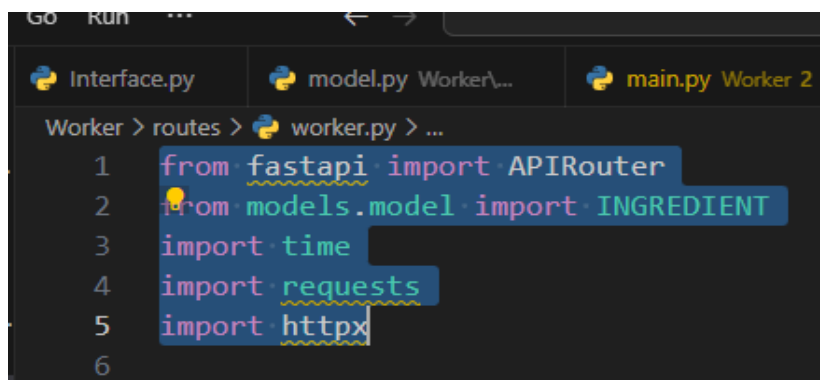
Cette condition vérifie si le fichier est exécuté directement (et non importé en tant que module). Si c'est le cas, le serveur uvicorn est lancé pour exécuter l'application. La fonction `uvicorn.run()` prend l'instance de l'application (app), l'adresse d'écoute (host) "localhost" et le port "8000" comme arguments pour démarrer le serveur **ASGI**.

### ➤ Descriptions des Endpoints et des méthodes du worker

#### • Explication du code worker.py

Ce programme exécute toutes les tâches confiées par le master et implémente les méthodes `send_result` et `get_task` qui permettent respectivement d'envoyer un résultat au serveur maître et de récupérer les tâches.

On commence d'abord par importer les modules du Framework FastAPI pour créer un routeur d'API.



```
1 from fastapi import APIRouter
2 from models.model import INGREDIENT
3 import time
4 import requests
5 import httpx
6
```

Par la suite nous avons le code ci-dessous :

```
# Worker's name
nom_workers = "Worker_Franglish"

# Create a FastAPI router
router = APIRouter()

# URL of the master server (where tasks are manage)
master_url = "http://192.168.10.48"
```

- ❖ **nom\_workers = " Worker\_Franglish "** : Cette ligne crée une variable `nom_workers` et lui attribue la valeur "Worker\_Franglish". C'est le nom que l'on attribue à notre worker.
- ❖ **router = APIRouter()** : Cette ligne crée une instance de la classe `APIRouter` fournie par FastAPI. Ce routeur sera utilisé pour définir les endpoints spécifiques à ce travailleur.
- ❖ **master\_url = "http://192.168.10.48"** : Cette ligne définit la variable `master_url` avec la valeur "http://192.168.10.48". Cette URL représente l'emplacement du serveur maître où les tâches sont gérées.

Par la suite nous allons définir une fonction **prepared\_fruit**

```
# Simulated fruit preparation function
def prepared_fruit(id_, fruit, t):
    time.sleep(t)
    return f"{id_} {fruit} prepared in {t} seconds"
```

Cette fonction simule **la préparation d'un fruit** en utilisant des paramètres fournis.

**id\_** : C'est un identifiant ou un numéro associé au fruit. Il peut être utilisé pour suivre ou identifier le fruit traité dans le contexte de l'application.

**fruit** : C'est le nom du fruit qui est en train d'être préparé.

**t** : C'est le temps en secondes que la fonction attend avant de simuler que la préparation du fruit est terminée. Pendant ce temps, la fonction fait une pause en utilisant **time.sleep(t)** pour simuler le processus de préparation.

Nous poursuivons notre code par ceci :

```

# Function to send a result to the master server
def send_result(url, data):
    with httpx.Client() as session:
        session.post(url, json=data)

# Function to get a task from the master server
def get_task():
    response = requests.post(f"{master_url}/send_task", json={"nom_worker": nom_workers})
    if response.status_code == 200:
        task = response.json()
        return task
    else:
        return None

```

Ces deux fonctions sont probablement utilisées pour la communication entre un serveur maître et des travailleurs (ou clients) dans un système distribué. Voici ce que font ces fonctions :

- ❖ **send\_result(url, data)** : Cette fonction envoie les résultats d'une tâche ou d'un traitement effectué par un travailleur au serveur maître en utilisant une requête HTTP POST. Les paramètres sont les suivants :
  - ❖ **url**: L'URL du serveur maître où les résultats doivent être envoyés.
  - ❖ **data**: Les données à envoyer, généralement au format JSON.
- ❖ **get\_task()** : Cette fonction est utilisée par un travailleur pour demander une nouvelle tâche au serveur maître. Elle envoie une requête HTTP POST au serveur maître pour obtenir une tâche à effectuer. Les paramètres sont les suivants :
  - ❖ **master\_url**: L'URL du serveur maître à contacter pour obtenir une tâche.
  - ❖ **nom\_workers**: Une donnée (probablement le nom du travailleur) à inclure dans la requête JSON.

Si la réponse du serveur maître a un code d'état **HTTP 200 (OK)**, la tâche est extraite du contenu JSON de la réponse et renvoyée. Sinon, si le code d'état indique un problème (**non 200**), la fonction renvoie **None** pour indiquer qu'aucune tâche n'est disponible ou qu'il y a eu une erreur dans la communication avec le serveur maître.

On poursuit notre implémentation du worker à travers ceci :



```

# Function to execute a task
def do_task():
    while True:
        task_data = get_task()
        while task_data:
            task = INGREDIENT(**task_data) # Convert the task data to an INGREDIENT object

            prepare_fruit = prepared_fruit(task.id, task.fruit, task.execution_time)
            print("Result sent to master:", prepare_fruit)
            url = f"{master_url}/result"
            data = {
                "task": {
                    "id": task.id,
                    "fruit": task.fruit,
                    "execution_time": task.execution_time,
                    "nom_worker": nom_workers
                },
                "result": prepare_fruit
            }
            send_result(url, data)

            task_data = get_task()

        time.sleep(3)
        print("No task available. Waiting...")

```

Cette fonction, **do\_task()**, est destinée à être exécutée par un travailleur dans un système distribué. Elle implémente la logique d'exécution des tâches. Voici ce que fait cette fonction :

- ❖ **Boucle Principale (while True):** Cette boucle s'exécute en continu, ce qui signifie que le travailleur reste actif en attendant de recevoir des tâches à exécuter depuis le serveur maître.
- ❖ **Obtention et Exécution des Tâches (task\_data = get\_task()):** À l'intérieur de la boucle principale, la fonction appelle la fonction `get_task()` pour obtenir une nouvelle tâche depuis le serveur maître. Si une tâche est disponible (`task_data` n'est pas `None`), alors une sous-boucle s'exécute pour traiter toutes les tâches disponibles.
- ❖ **Conversion de Données de Tâche (task = INGREDIENT(\*\*task\_data)):** La donnée de tâche obtenue du serveur maître est utilisée pour créer un objet de type `INGREDIENT` qui contient les détails de la tâche à exécuter.
- ❖ **Exécution de la Tâche de Préparation du Fruit (prepare\_fruit = prepared\_fruit(...)) :** La fonction appelle la fonction `prepared_fruit()` (que vous avez mentionnée dans une question précédente) pour simuler la préparation d'un fruit en utilisant les détails de la tâche. Le résultat de la préparation est stocké dans la variable `prepare_fruit`.

- ❖ **Envoi du Résultat au Serveur Maître** : Les détails de la tâche exécutée et le résultat de la préparation sont préparés dans un format approprié et envoyés au serveur maître en utilisant la fonction `send_result()`.
- ❖ **Répétition du Processus (`task_data = get_task()`)**: Après avoir traité une tâche, la fonction appelle à nouveau `get_task()` pour vérifier s'il y a d'autres tâches à traiter. Si oui, le processus se répète pour chaque tâche disponible.
- ❖ **Attente en L'absence de Tâches (`time.sleep(3)`)**: Si aucune tâche n'est disponible, la fonction attend pendant 3 secondes avant de vérifier à nouveau la disponibilité des tâches. Cela évite que le travailleur ne vérifie en permanence s'il y a des tâches à exécuter.

Par la suite nous obtenons ceci :

```
# Add a docstring for the `send_result` function
def send_result(url, data):
    """
    Sends a result to the master server.

    Parameters:
    - url (str): The URL to send the result to.
    - data (dict): The result data to send.

    Returns:
    None
    """
    with httpx.Client() as session:
        session.post(url, json=data)
```

Nous ajoutons une docstring à la `send_result` fonction.

Nous terminons le code avec ceci :

```
def do_task():
    """
    Continuously fetches and performs tasks.

    Returns:
    None
    """
    while True:
        task_data = get_task()
        while task_data:
            task = INGREDIENT(**task_data) # Convert the task data to an INGREDIENT object

            prepare_fruit = prepared_fruit(task.id, task.fruit, task.execution_time)
            print("Result sent to master:", prepare_fruit)
            url = f"{master_url}/result"
            data = {
                "task": {
                    "id": task.id,
                    "fruit": task.fruit,
                    "execution_time": task.execution_time,
                    "nom_worker": nom_workers
                },
                "result": prepare_fruit
            }
            send_result(url, data)
            task_data = get_task()
```

```
time.sleep(3)
print("No task available. Waiting...")
```

La fonction **do\_task()** effectue la gestion continue et l'exécution des tâches provenant du serveur maître dans un environnement distribué. Voici ce que fait cette fonction :

- ❖ **Boucle Principale (while True):** Cette boucle s'exécute en continu, ce qui signifie que le travailleur reste actif et prêt à effectuer des tâches en permanence.
- ❖ **Obtention et Exécution des Tâches (task\_data = get\_task()):** À l'intérieur de la boucle principale, la fonction appelle la fonction `get_task()` pour obtenir une nouvelle tâche depuis le serveur maître. Si une tâche est disponible (`task_data` n'est pas `None`), alors une sous-boucle s'exécute pour traiter toutes les tâches disponibles.
- ❖ **Conversion de Données de Tâche (task = INGREDIENT(\*\*task\_data)):** La donnée de tâche obtenue du serveur maître est utilisée pour créer un objet de type `INGREDIENT` (c'est un terme générique dans votre code) qui contient les détails de la tâche à exécuter.
- ❖ **Exécution de la Tâche de Préparation du Fruit (prepare\_fruit = prepared\_fruit(...)) :** La fonction appelle la fonction `prepared_fruit()` pour simuler la préparation d'un fruit en utilisant les détails de la tâche. Le résultat de la préparation est stocké dans la variable `prepare_fruit`.
- ❖ **Envoi du Résultat au Serveur Maître :** Les détails de la tâche exécutée et le résultat de la préparation sont préparés dans un format approprié et envoyés au serveur maître en utilisant la fonction `send_result()`.
- ❖ **Répétition du Processus (task\_data = get\_task()):** Après avoir traité une tâche, la fonction appelle à nouveau `get_task()` pour vérifier s'il y a d'autres tâches à traiter. Si oui, le processus se répète pour chaque tâche disponible.
- ❖ **Attente en L'absence de Tâches (time.sleep(3)):** Si aucune tâche n'est disponible, la fonction attend pendant 3 secondes avant de vérifier à nouveau la disponibilité des tâches. Cela évite que le travailleur ne vérifie en permanence s'il y a des tâches à exécuter.

## ➤ Descriptions des Endpoints et des méthodes du shared

Le “**Shared**” est l'API qui se charge de recevoir les ingrédients de la salade de fruit du client et ensuite l'envoi au Master.

- Méthode `create_table()`

```
router = APIRouter()

db_path = "fruits.db"

Test this function
def create_table():
    connection = sqlite3.connect(db_path)
    cursor = connection.cursor()
    cursor.execute("DROP TABLE IF EXISTS fruits")
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS fruits (
            id INTEGER PRIMARY KEY,
            name TEXT NOT NULL
        )
    """)
    connection.commit()
    connection.close()
```

Cette fonction est appelée dans l'événement de démarrage de FastAPI (`router.on_event("startup")`). Elle crée une table nommée "fruits" dans la base de données SQLite. Cette table a deux colonnes : "id" (une clé primaire auto-incrémentée) et "name" (qui stocke le nom du fruit).

- La méthode `startup_event()`

```
Test this function
25 @router.on_event("startup")
26 async def startup_event():
27     create_table()
28
```

La fonction `startup_event()` appelle la fonction `create_table()` pour s'assurer que la table "fruits" existe dans la base de données lorsque l'application démarre. En d'autres termes, chaque fois que vous démarrez l'application, la fonction `create_table()` est appelée pour créer la table si elle n'existe pas déjà.

- La méthode `create_task()`

```
Test this function
29 @router.post("/create_task/", response_model=List[Fruit])
30 async def create_task(liste_fruits: List[Fruit]):
31     """
32     Créer une tâche à exécuter. Il s'agit d'une liste de fruits à traiter.
33
34     Paramètres:
35     - liste_fruits (List[Fruit]): liste de fruits à traiter
36
37     """
38     connection = sqlite3.connect(db_path)
39     cursor = connection.cursor()
40
41     for fruit in liste_fruits:
42         cursor.execute("INSERT INTO fruits (name) VALUES (?)", (fruit.fruit,))
43
44     connection.commit()
45     connection.close()
46
47     return JSONResponse(status_code=200, content={"message": "Task created"})
48
```

Cette méthode POST permet de créer une tâche à exécuter en insérant les noms de fruits fournis dans la liste `liste_fruits` dans la table "fruits" de la base de données. Chaque fruit est inséré en tant que nouvel enregistrement dans la table. La méthode renvoie une réponse JSON indiquant que la tâche a été créée. Elle est utilisée par le 'Client' pour créer la liste de fruits à préparer.

- **La méthode `get_tasks()`**

```
Test this function
49 @router.get("/get_tasks_to_do/", response_model=List[Fruit])
50 async def get_tasks():
51     """
52     Récupère toutes les tâches à exécuter.
53     """
54
55     connection = sqlite3.connect(db_path)
56     cursor = connection.cursor()
57     cursor.execute("SELECT name FROM fruits")
58     fruits = cursor.fetchall()
59     # Convertir la liste de tuples en liste de Fruit
60     fruits = [{"fruit": fruit[0]} for fruit in fruits]
61
62
63     cursor.execute("DELETE FROM fruits")
64     connection.commit()
65
66     connection.close()
67     if len(fruits) == 0:
68         return JSONResponse(content=fruits, status_code=202)
69     else:
70         return JSONResponse(content=fruits, status_code=200)
```

Cette méthode GET récupère toutes les tâches à exécuter depuis la table "fruits" de la base de données. Elle sélectionne tous les noms de fruits enregistrés dans la table, les stocke sous forme de listes de tuples, puis les convertit en une liste de dictionnaires où chaque dictionnaire représente un fruit. Ensuite, elle supprime toutes les lignes de la table "fruits" pour vider la liste des tâches à exécuter. Si la liste de fruits récupérée est vide, la méthode renvoie une réponse JSON avec le code de statut 202 (Accepted), sinon elle renvoie une réponse JSON avec le code de statut 200 (OK). Cette méthode est appelée par le 'Master' pour récupérer les fruits créés par le client et ensuite crée les tâches qui seront exécuter par les workers.

## Conclusion

En clôture de ce projet au sein du cours sur les Bases des Systèmes Distribués, nous avons franchi une étape significative en explorant les aspects pratiques de la distribution de données et de tâches au sein de réseaux interconnectés. Notre objectif d'implémenter une exécution répartie en utilisant le framework FastAPI a renforcé nos compétences dans la conception et la réalisation de systèmes distribués modernes.

L'adoption du framework FastAPI s'est avérée particulièrement pertinente, offrant une approche moderne et performante pour la création de services web REST. En effectuant la transition d'un modèle de communication RPC vers des services web basés sur HTTP, nous avons non seulement amélioré la flexibilité et la communication entre les différentes parties du système, mais nous avons également mis en lumière les avantages en termes d'interopérabilité et d'extensibilité qu'offre FastAPI. Au fil de ce projet, l'architecture maître/ouvrier dans un contexte distribué, avec FastAPI comme mécanisme de communication, nous a permis de saisir plus concrètement les enjeux complexes des systèmes distribués. En appréhendant les défis de coordination, de sécurité et de gestion des ressources, nous avons gagné une compréhension approfondie des principes fondamentaux qui régissent ces systèmes.

En résumé, notre travail sur ce projet a renforcé notre apprentissage des systèmes distribués tout en nous exposant à un outil moderne et puissant, à savoir le framework FastAPI. Les compétences acquises tout au long de ce projet nous seront inestimables alors que nous nous efforçons de relever les défis toujours changeants et passionnants du domaine des systèmes distribués. Notre expérience avec FastAPI restera une pierre angulaire de notre progression continue dans la création de systèmes distribués innovants et performants.