

Microservices

Version 1.2 (2024)
from V1.0 Sami
Souihi

Tiphaine Henry

En Brief:

Dans ce cours, nous explorerons les concepts fondamentaux des microservices, leurs avantages par rapport aux architectures monolithiques, et comment utiliser Python et Flask pour les implémenter efficacement.

Objectifs :

- Comprendre les fondamentaux
- Savoir mettre en place des microservices
- Savoir deployer des microservices

Evaluation :

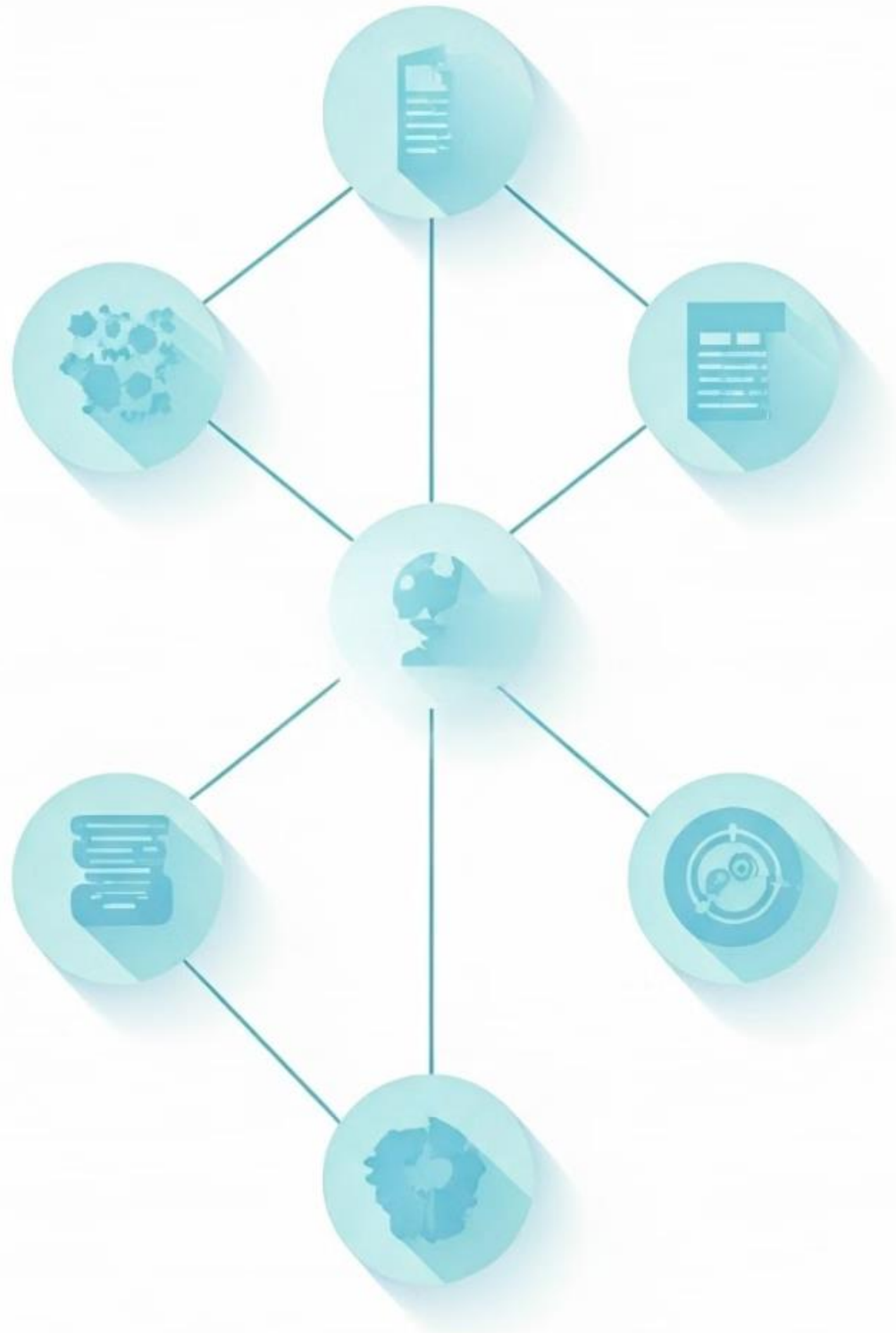
- Controle continu 50%
- Examen 50%

Microservices vs API : Comprendre la Différence

Les microservices et les API gagnent en popularité dans le domaine du développement logiciel, répondant au besoin croissant de créer des applications évolutives, sécurisées et flexibles dans des délais plus courts. Les demandes des clients évoluent rapidement, exigeant des solutions logicielles qui simplifient leurs tâches et offrent plus de commodité.

Alors, quelle est la meilleure approche pour construire et déployer votre application?





Qu'est-ce que les Microservices ?

Définition

Les microservices sont des services plus petits et faiblement couplés que l'on peut déployer indépendamment. Chaque service représente une fonction spécifique de l'application.

Architecture

Dans une architecture de microservices, les fonctions d'une application sont divisées en composants plus petits ayant des objectifs spécifiques, avec leurs propres piles technologiques et méthodes de gestion des données.

Communication

Les services peuvent communiquer entre eux via des API REST, des brokers de messages ou encore du streaming.



Avantages des Microservices

1 Scalabilité

Chaque service peut être mis à l'échelle indépendamment en fonction de sa charge de travail, optimisant ainsi l'utilisation des ressources. Par exemple, le service de recherche d'une application e-commerce peut être mis à l'échelle sans affecter les autres services.

2 Déploiement Indépendant

Les microservices permettent des mises à jour fréquentes et isolées, minimisant les temps d'arrêt. Lorsqu'un microservice est mis à jour, les autres continuent de fonctionner sans interruption, assurant une haute disponibilité de l'application.

3 Technologie Variée

Dans une architecture de microservices, chaque service peut être construit en utilisant la technologie la plus adaptée à ses besoins spécifiques, offrant ainsi une grande flexibilité dans les choix de développement.

Comparaison avec les Architectures Monolithiques

Architecture Monolithique

Une application unique qui regroupe toutes les fonctionnalités. Simple pour des petites applications mais problématique pour les applications de grande envergure. La moindre mise à jour nécessite un redéploiement de l'ensemble de l'application.

Architecture Microservices

Modularité et indépendance des composants. Chaque service peut être modifié et déployé sans affecter les autres, offrant une flexibilité accrue et permettant une scalabilité horizontale et verticale. Amélioration de la résilience : si un service échoue, les autres peuvent continuer de fonctionner normalement.

Composants des Microservices



Types de Microservices

Microservices sans état

Ne maintiennent pas d'état de session entre les requêtes. Ils sont les éléments de base des systèmes distribués. Même si une instance de service est supprimée, la logique de traitement globale du service n'est pas affectée.

Microservices avec état

Maintiennent ou stockent des états de session ou des données dans le code. Les microservices qui communiquent entre eux maintiennent toujours les requêtes de service.

Plan du cours :

- 1) Introduction aux APIs
- 2) Gestion de fichiers structurés
- 3) Gestion de bases de données
- 4) Interface utilisateur / API

Plan du cours :

- 1) Introduction aux APIs
- 2) Gestion de fichiers semi-structurés
- 3) Gestion de bases de données
- 4) Interface utilisateur / API

Docteur, c'est quoi une API?

- Dans un restaurant, on vous remet un menu avec les articles que vous pouvez commander.
- Vous pouvez aussi demander au serveur certaines adaptations (cuisson de la viande, une allergie,)
- Le menu représente la liste d'action possible et le serveur est votre API ☺



APIs (Application Programming Interfaces)

Une API permet d'exposer des données ou des fonctionnalités d'une application afin que d'autres applications les utilisent.



Une API, ça sert à quoi?



**LES APIS POUR MANIPULER
DES DOCUMENTS**



**LES APIS POUR RÉCUPÉRER
DES DONNÉES**



**LES APIS POUR MANIPULER
DES OBJETS MULTIMEDIA**



**LES APIS POUR MANIPULER
DES PÉRIPHÉRIQUES**

Qu'est-ce qu'une API ?

1 Définition

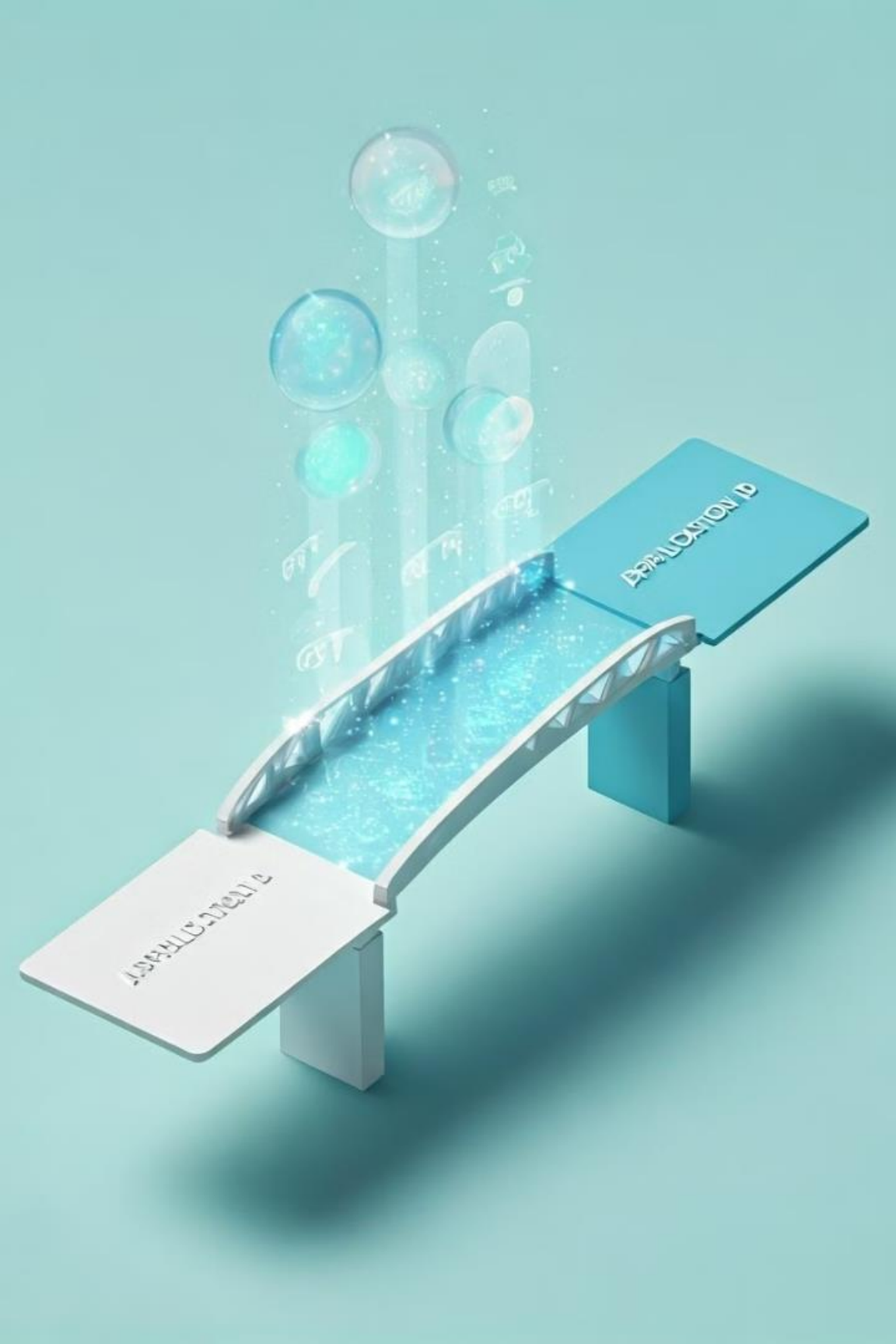
Une Interface de Programmation d'Application (API) est un intermédiaire logiciel entre deux applications qui interagissent entre elles. Elle connecte deux ordinateurs ou programmes informatiques via une interface.

2 Fonction

Les API simplifient la programmation en cachant les détails internes d'un système et en exposant les parties utiles pour un programmeur, tout en maintenant la cohérence malgré les changements internes.

3 Types de requêtes

Les API utilisent quatre types de requêtes : GET, PUT, DELETE, POST.



Composants d'une API



Protocoles

Ensemble de règles définissant la manière dont les applications interagissent, comme HTTP, SOAP, XML-RPC, REST.



Format

Style d'échange de données entre applications, définissant comment l'API récupère et fournit les données aux consommateurs.



Procédures

Tâches ou fonctions spécifiques qu'une application exécute.



Outils

Utilisés pour construire, tester et gérer les API, comme AWS, IBM Cloud, SoapUI, JMeter.

INFOGRAPHICS



Types d'API

Publiques

Disponibles pour tout utilisateur tiers

Privées

Conçues pour améliorer les services au sein d'une entreprise

Partenaires

Partagées uniquement avec les partenaires commerciaux

Web

Fournissent des fonctionnalités et des transferts de données entre services web

Système d'exploitation

Définissent comment une application peut utiliser les services d'un OS

Avantages des Microservices

1

Modularité

Divise les services en modules distincts, facilitant le développement et les tests.

2

Développement distribué

Permet à de petites équipes de développer, tester et déployer des services en parallèle.

3

Évolutivité

Permet de mettre à l'échelle uniquement les composants nécessaires.

4

Déploiement indépendant

Les changements peuvent être apportés sans affecter l'ensemble de l'application.





Avantages des API

Vitesse

Les API offrent une vitesse incroyable pour diverses tâches, accélérant les opérations et réduisant les tracas pour les clients.

Évolutivité

Offre une flexibilité pour étendre les produits, augmenter les catalogues et gérer la croissance des données.

Sécurité

Améliore la sécurité de l'application en agissant comme intermédiaire entre le client et le serveur.

Innovation

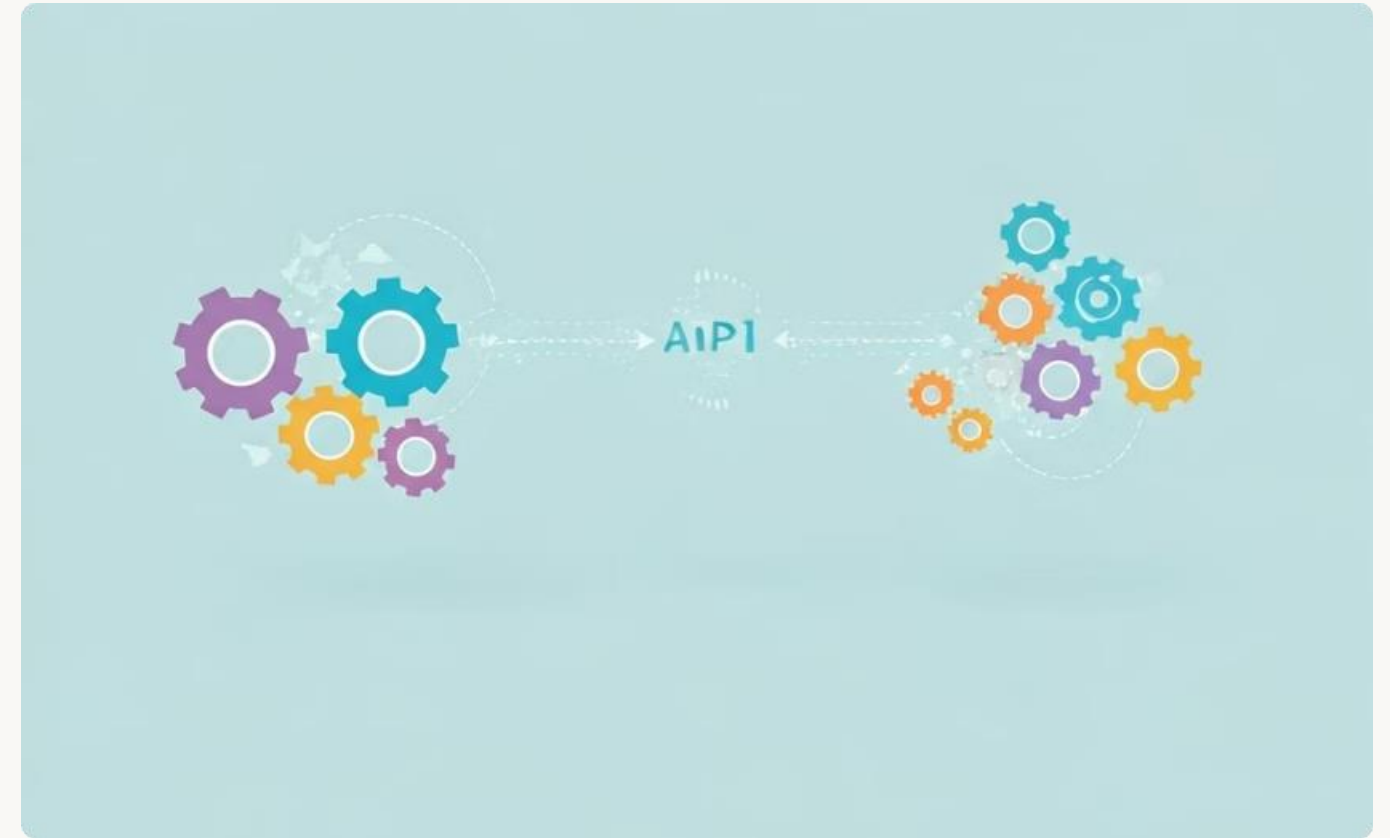
Favorise l'innovation en permettant l'intégration rapide de nouvelles fonctionnalités.

Microservices et API : Travailler ensemble



Intégration

Les microservices et les API peuvent travailler ensemble efficacement. Les API facilitent l'intégration et la gestion des microservices, permettant à cette nouvelle architecture de coexister avec les systèmes traditionnels.



Synergie

En combinant les microservices avec les API, les organisations peuvent bénéficier de tous les avantages des microservices tout en limitant leurs inconvénients, offrant ainsi une solution puissante et flexible pour le développement d'applications modernes.

Python et Flask pour les Microservices

Python est un langage de programmation polyvalent et largement adopté dans l'industrie, offrant une grande simplicité de syntaxe et une courbe d'apprentissage douce. Flask est un microframework Python qui permet de créer des API REST de manière rapide et efficace. Il est léger, flexible, et particulièrement adapté à une architecture de microservices où chaque service doit rester simple et indépendant.



Python

Langage polyvalent et simple



Flask

Microframework léger et flexible



API REST

Création rapide et efficace



Avantages de Flask dans les Microservices

Léger et Rapide

Flask ne nécessite que peu de configuration initiale, ce qui le rend idéal pour développer rapidement des microservices indépendants.

Flexibilité

Grâce à sa nature minimaliste, Flask permet aux développeurs d'ajouter uniquement les fonctionnalités dont ils ont besoin.

Communauté et Support

Flask dispose d'une large communauté d'utilisateurs et de nombreuses extensions pour ajouter des fonctionnalités.



Configuration de l'Environnement de Développement

Pour démarrer un projet Flask, il est recommandé d'utiliser un environnement virtuel, qui permet d'isoler les dépendances spécifiques du projet. Cela est particulièrement important dans une architecture microservices où chaque service peut nécessiter différentes bibliothèques ou versions de bibliothèques.

1

Création de l'Environnement Virtuel

Utilisez la commande : `python3 -m venv env`

2

Activation de l'Environnement

Exécutez : `source env/bin/activate`

3

Installation de Flask

Installez Flask avec : `pip install Flask`



Structure de Base d'un Microservice Flask

Dans une architecture microservices, chaque service devrait idéalement être dans son propre dossier avec sa configuration spécifique pour faciliter les déploiements et les tests. Voici une structure de base pour un microservice avec Flask :

```
/service-utilisateur
├─ app.py           # Fichier principal qui contient la logique de l'API
├─ requirements.txt # Liste des dépendances Python
└─ config.py        # Fichier de configuration (e.g., gestion de clés, identifiants, mode débogage)
```


Création d'une API REST de Base avec Flask

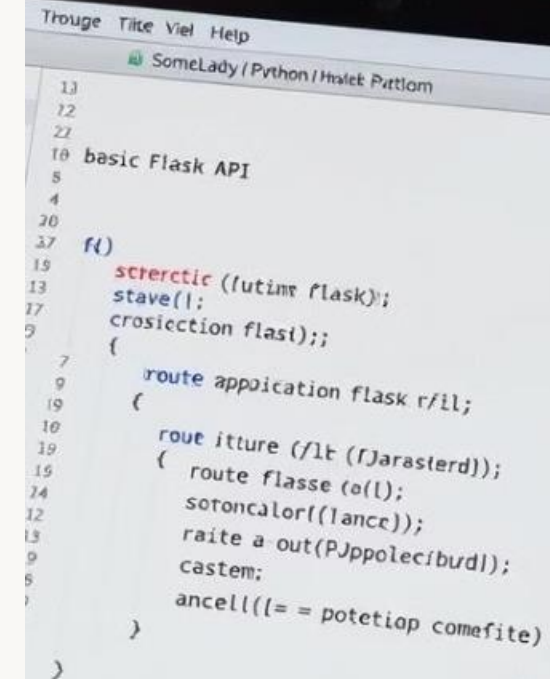
L'exemple ci-dessous montre comment créer une API REST de base avec Flask. Ce code définit une route simple qui renvoie un message de bienvenue en format JSON. Cette API de base est accessible via une requête HTTP GET, ce qui est souvent le point de départ pour un microservice simple.

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/salutation', methods=['GET'])
def salutation():
    return jsonify(message="Bonjour, bienvenue dans notre API de microservices !")

if __name__ == '__main__':
    app.run(debug=True)
```



```
13
12
22
10 basic Flask API
8
4
20
27 f()
19   stercitc (future flask);
13   stave(!;
17   crosiection flas!);;
9   {
7       route appoication flask r/il;
9       {
19         route ittute (/lt (/Jarasterd));
19         route flasce (o(!;
24         sotoncalor(!lance));
12         raite a out(PJppolecibudl);
13         castem;
9         ancell(!= = potetiap comebite)
6       }
1     }
```

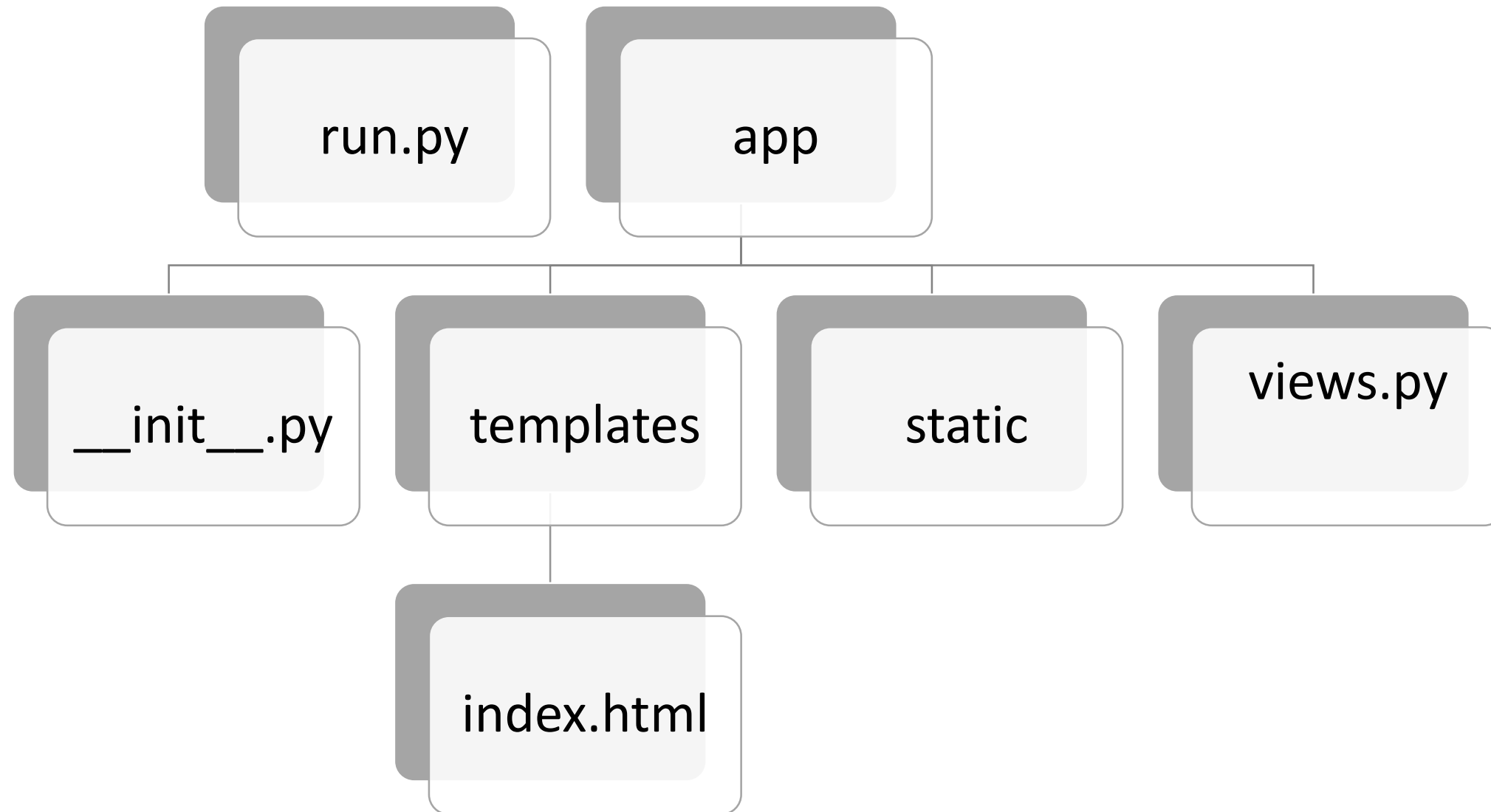
Extension de l'API avec Différentes Méthodes HTTP

Pour étendre cette API, nous pouvons ajouter d'autres routes en utilisant différentes méthodes HTTP, comme POST pour créer une nouvelle ressource ou DELETE pour en supprimer une. Voici un exemple avec une route POST pour ajouter un utilisateur :

```
@app.route('/api/utilisateurs', methods=['POST'])
def ajouter_utilisateur():
    data = request.get_json()
    nom = data.get('nom')
    return jsonify(message=f"Utilisateur {nom} ajouté avec succès!"), 201
```

Dans cet exemple, la fonction `ajouter_utilisateur` reçoit des données JSON envoyées par le client et extrait le nom de l'utilisateur. Le serveur renvoie une réponse JSON confirmant l'ajout de l'utilisateur avec un code d'état HTTP 201 (Créé), indiquant que l'opération a réussi.

Structure d'un microservice un peu complexe (MVC vs WebAPI)



run.py

```
from app import app  
app.run(debug = True)
```

Ce fichier ne sert qu'au lancement du serveur web ainsi il vous est possible de passer du mode debug au mode prod sans altérer le code de l'application (et même de changer de serveur d'application)

__init__.py

```
from flask import Flask  
app= Flask(__name__)  
from app import views
```

Il s'agit du point d'entrée de votre application

views.py

```
from app import app
@app.route('/')
def index():
    return "hello world"
```

Il s'agit de votre contrôleur. Concrètement, vous définissez une route '/' (ç-à-d un url : http://@IP/ ou http://nom_domaine/). Lors de l'appel de cette route, la fonction index est exécuté.

Allons plus loin

- Et si au lieu d'afficher une simple chaîne de caractère nous affichons une page web????

views.py

```
from app import app
from flask import render_template
@app.route('/')
def index():
    return render_template('index.html')
```


Passage de paramètres

views.py

```
from app import app
from flask import
render_template
@app.route('/')
def index():
    user={'name':'john',
'surname':'doe'}
    return render_template
('index.html', title='MDM',
utilisateur=user)
```

Index.html

```
<html>
<head>
    <title> {{ title }}
</title>
</head>
<body>
    hello
    {{utilisateur.name }}
    {{utilisateur.surname}}
</body>
</html>
```

Et si vous me faites l'inverse?

À faire C.C.

- Faire en sorte d'envoyer des informations à une fonction du contrôleur via des paramètres en get (transmettre des paramètres via l'url)

`http://127.0.0.1:5000/params?surname=Dupont&name=Jean`

Plan du cours :

- 1) Introduction aux APIs
- 2) Gestion de fichiers semi-structurés
- 3) Gestion de bases de données
- 4) Interface utilisateur / API

Définition

Les données semi-structurées sont des données qui ne sont pas conformes aux normes des données structurées classiques, mais qui contiennent des balises ou d'autres types de balisage qui identifient des entités individuelles et distinctes dans les données.

Objectifs



DÉCOUVERTE DE JSON



MANIPULER DU JSON EN
PYTHON



INTÉGRATION JSON & FLASK

Exemple de JSON

- "JSON" signifie "JavaScript Object Notation"
- Malgré son nom, JSON est indépendante du langage JS.
- Il permet de décrire des objets en tant que paires clé-valeur

```
{  
  "type": "Feature",  
  "id": "63d55408-39a1-4284-b413-9afb1aa86b52",  
  "geometry": {  
    "type": "Point",  
    "coordinates": [  
      24.93218,  
      60.19897  
    ]  
  }  
}
```

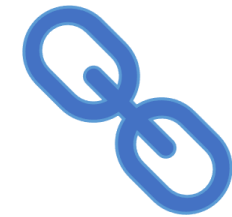


Syntaxe JSON

- Un objet est un ensemble non ordonné de paires nom/valeur :
 - Les paires sont encadrées par des accolades { }
 - Le nom et la valeur sont séparés par deux points
 - Les paires sont séparées par des virgules
 - Exemple : { "nom" : "John Doe", " age" : 5 }
- Un tableau est une collection ordonnée de valeurs
 - Les valeurs sont placées entre parenthèses, []
 - Les valeurs sont séparées par des virgules
 - Exemple : ["html", "xml", "css"]

Syntaxe JSON

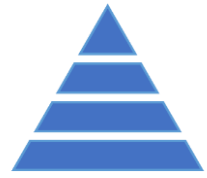
- Une valeur peut être : Une chaîne de caractères, un nombre, un booléen, nul, un objet ou un tableau
 - Les valeurs peuvent être imbriquées
- Les chaînes de caractères sont entre guillemets et peuvent contenir l'assortiment habituel de caractères échappés
- Les nombres ont la syntaxe habituelle C/C++/Java, y compris la notation exponentielle (E)
 - Tous les nombres sont décimaux



EVAL

La méthode `eval(string)` compile et exécute la chaîne donnée

- La chaîne peut être une expression, une déclaration ou une séquence de déclarations
- Les expressions peuvent inclure des variables et des propriétés d'objet
- `eval` renvoie la valeur de la dernière expression évaluée - Lorsqu'il est appliqué à JSON, `eval` renvoie l'objet décrit



Comparaison entre JSON et XML

Similarités :

- Les deux sont lisibles par l'homme
- Les deux ont une syntaxe très simple
- Les deux sont hiérarchiques
- Les deux sont indépendants du langage

Différences :


- La syntaxe est différente
- JSON est moins verbeux
- JSON inclut des tableaux
- Les noms dans JSON ne doivent pas être des mots réservés par JavaScript

YAML

- YAML peut être un acronyme pour :
 - Yet Another Markup Language
 - YAML Ain't Markup Language
- Comme JSON, le but de YAML est de représenter des types de données typiques en notation lisible par l'homme
- YAML est techniquement un sur-ensemble de JSON, avec beaucoup plus de capacités (listes, casting, etc.)
- **YAML - Lorsque JSON ne suffit pas, considérez YAML**

Encodage et décodage JSON

Le module « json » fournit une API familière aux utilisateurs des modules marshal et pickle de la bibliothèque standard.



Les encodeurs et décodeurs de ce module conservent l'ordre d'entrée et de sortie par défaut. L'ordre n'est perdu que si les conteneurs sous-jacents ne sont pas ordonnés.

- Avant Python 3.7, dict n'était pas garanti d'être ordonné, donc les entrées et sorties étaient généralement mélangées à moins d'utiliser explicitement un `collections.OrderedDict`.
- À partir de Python 3.7, un dict conserve son ordre, il n'est donc plus nécessaire d'utiliser un `collections.OrderedDict` pour générer et analyser du JSON.

Sérialisation de JSON

- Que se passe-t-il lorsqu'un ordinateur traite un grand nombre d'informations ?
- La bibliothèque json expose la méthode dump() pour écrire des données dans des fichiers. Il existe également une méthode dumps() (prononcée "dump-s") pour écrire dans une chaîne de caractères Python.
- Les objets Python simples sont traduits en JSON selon une conversion assez intuitive.

Conversions

- source:

<https://docs.python.org/3/library/json.html#encoders-and-decoders>

JSON

object

array

string

number (int)

number (real)

true

false

null

Python

dict

list

str

int

float

True

False

None

JSON Read

```
{
  "type": "FeatureCollection",
  "features": [ {
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [42.0, 21.0]
    },
    "properties": {
      "prop0": "value0"
    }
  }
]
```

data.json

```
import json
```

```
f = open('data.json')
```

```
data = json.load(f)
```

```
f.close()
```

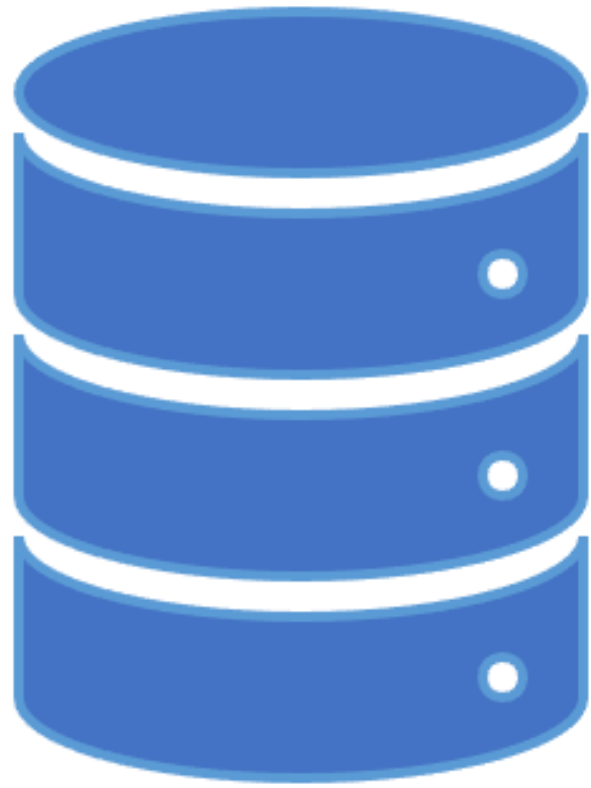
```
print(data)
```

```
print(data["features"])
```

```
print(data["features"][0]["geometry"])
```

```
for i in data["features"]:
```

```
    print(i["geometry"]["coordinates"][0])
```

JSON write

```
import json
```

```
f = open('data.json')
```

```
data = json.load(f)
```

```
f.close()
```

```
f = open('out.json', 'w')
```

```
json.dump(data, f)
```

```
f.close()
```

Formatted printing

```
print(json.dumps(data["features"], sort_keys=True, indent=4))
```

```
[  
  {  
    "geometry": {  
      "coordinates": [  
        42.0,  
        21.0  
      ],  
      "type": "Point"  
    },  
    "properties": {  
      "prop0": "value0"  
    },  
    "type": "Feature"  
  }  
]
```

Dumps offre une option de formatage d'impression agréable :

<https://docs.python.org/3/library/json.html>

Outil de contrôle de JSON

```
python -m json.tool < data.json
```

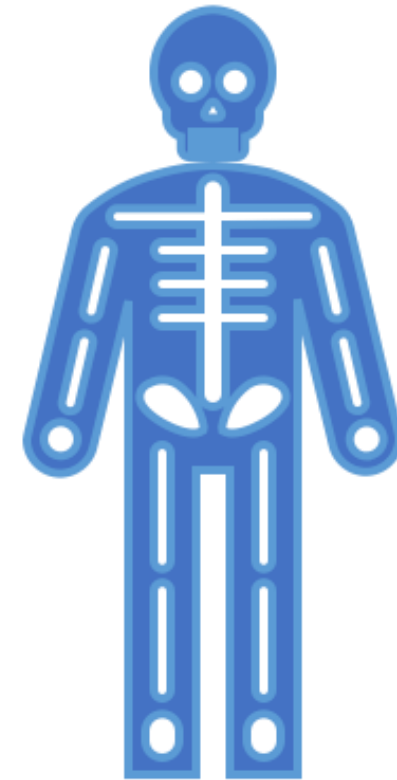
JSON et FLASK

- Flask possède un utilitaire appelé jsonify() qui facilite le retour des réponses JSON

```
from flask import Flask
from flask import jsonify
app = Flask(__name__)
@app.route('/api/get-json')
def hello():
    return jsonify(hello='world')
```

Exercice

Proposer une fonction permettant de chercher dans un dictionnaire les paramètres de santé d'une personne et de les renvoyer sous forme d'objet JSON





Principe de Responsabilité Unique

Le principe de responsabilité unique (SRP) est un concept clé dans la conception des microservices. Il stipule que chaque service doit avoir une seule responsabilité, centrée sur une tâche ou une fonction métier spécifique. Cette approche réduit le couplage entre les services et simplifie la maintenance.

Dans une application e-commerce, par exemple, on pourrait avoir un service dédié aux utilisateurs, un autre aux produits, et un troisième aux commandes, chacun avec ses responsabilités distinctes.

Service Utilisateurs

Gestion des profils, authentification et sécurité des utilisateurs.

Service Produits

Ajout, mise à jour et suppression des informations de produit.

Service Commandes

Gestion des transactions, paiements et logistique.



Conception Pilotée par le Domaine

La conception pilotée par le domaine (DDD) est une méthodologie qui organise les microservices en fonction des besoins métier. Elle divise l'application en 'domaines' représentant différentes zones fonctionnelles, chacun avec son propre modèle métier et sa logique.

Pour une application de gestion de bibliothèque, on pourrait avoir des domaines distincts pour les livres, les utilisateurs et les emprunts, chacun gérant ses propres aspects spécifiques du système.

Domaine des Livres

Gestion des informations relatives aux livres (titres, auteurs, disponibilité).

Domaine des Utilisateurs

Gestion des informations de profil des membres de la bibliothèque.

Domaine des Emprunts

Suivi des livres empruntés et des dates de retour.

Implémentation avec Flask

Flask est un framework populaire pour implémenter des microservices en Python. Voici un exemple de structure pour un service de gestion des produits dans une application e-commerce, illustrant l'application du SRP et du DDD :

```
from flask import Flask, jsonify, request
```

```
app = Flask(__name__)
```

```
@app.route('/api/produits', methods=['POST'])
```

```
def ajouter_produit():
```

```
    data = request.get_json()
```

```
    nom = data.get('nom')
```

```
    prix = data.get('prix')
```

```
    # Logique pour ajouter le produit en base de données
```

```
    return jsonify(message=f"Produit {nom} ajouté avec succès à {prix} EUR!"), 201
```

```
@app.route('/api/produits/', methods=['GET'])
```

```
def obtenir_produit(id):
```

```
    # Logique pour récupérer le produit depuis la base de données
```

```
    return jsonify(id=id, nom="Produit Ex.", prix=100)
```


Communication Synchrone entre Microservices

La communication synchrone dans une architecture de microservices utilise des appels directs d'API REST. Chaque service envoie une requête HTTP à un autre service et attend une réponse avant de continuer. Flask est couramment utilisé pour implémenter cette communication synchrone avec des endpoints REST.

Voici un exemple de code pour effectuer une requête HTTP d'un microservice vers un autre :

```
import requests

def obtenir_informations_utilisateur(user_id):
    response = requests.get(f"http://service-utilisateur/api/utilisateurs/{user_id}")
    if response.status_code == 200:
        return response.json()
    else:
        return {"error": "Utilisateur non trouvé"}
```

- 1

Requête

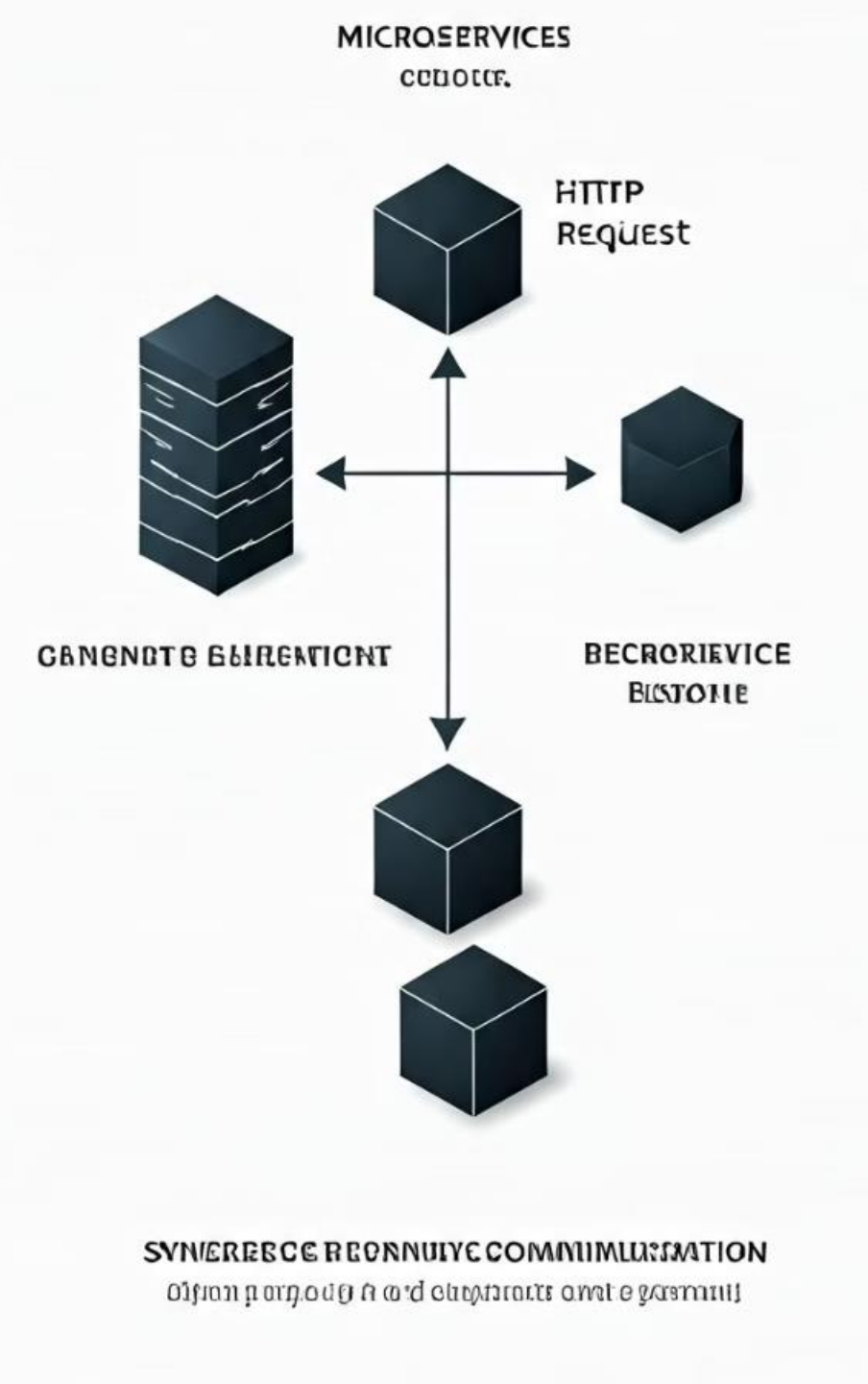
Le service envoie une requête HTTP
- 2

Attente

Le service attend la réponse
- 3

Réponse

Le service reçoit et traite la réponse



Communication Asynchrone avec les Files de Messages

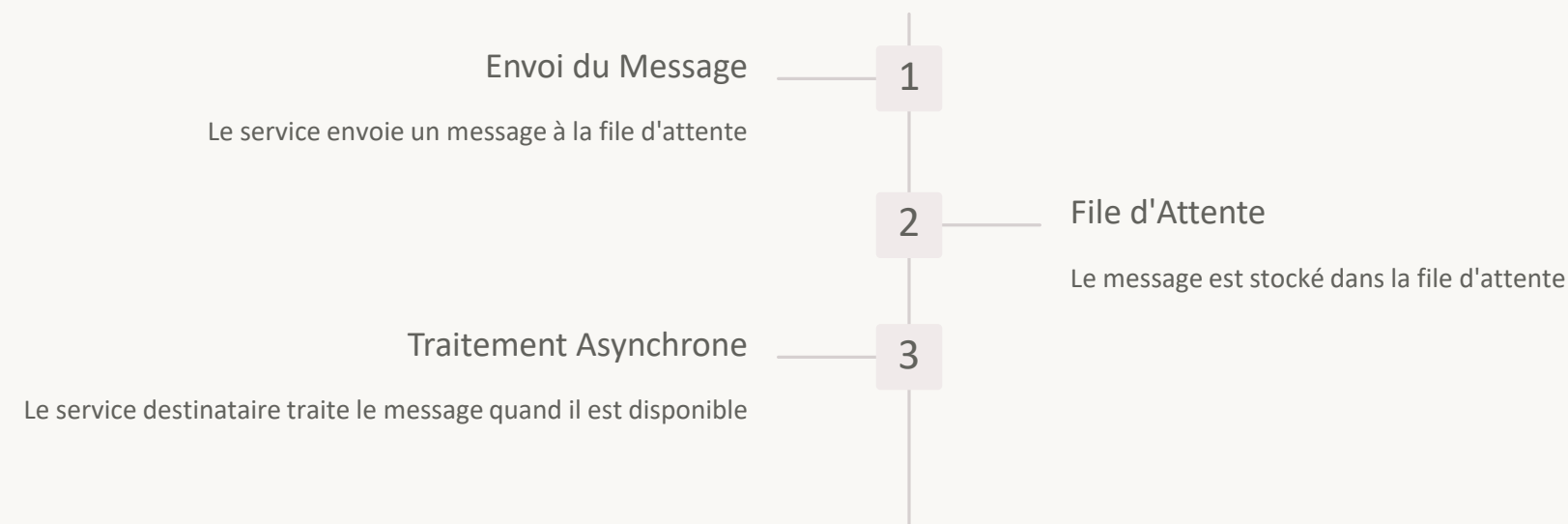
La communication asynchrone permet à un microservice d'envoyer un message à un autre sans attendre une réponse immédiate. Cela améliore la réactivité et la résilience du système. Cette approche utilise généralement des files de messages comme RabbitMQ ou Kafka.

Voici un exemple de code pour publier un message dans RabbitMQ :

```
import pika

def envoyer_message_commande(order_id):
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    channel = connection.channel()
    channel.queue_declare(queue='commandes')

    message = f"Nouvelle commande : {order_id}"
    channel.basic_publish(exchange='', routing_key='commandes', body=message)
    print("Message envoyé :", message)
    connection.close()
```



Gestion des Erreurs et Tolérance aux Pannes

La gestion des erreurs et la tolérance aux pannes sont cruciales dans une architecture de microservices. Des techniques comme le circuit breaker, les timeouts et les retries sont couramment utilisées pour gérer les défaillances potentielles des services.

Voici un exemple d'implémentation simple d'un retry en Python pour gérer les erreurs de connexion :

```
import requests
from time import sleep

def obtenir_donnees_service(url, retries=3, delay=2):
    for i in range(retries):
        try:
            response = requests.get(url)
            if response.status_code == 200:
                return response.json()
            else:
                print(f"Tentative {i+1} échouée.")
        except requests.ConnectionError:
            print(f"Connexion échouée. Tentative {i+1}/{retries}.")
            sleep(delay)
    return {"error": "Service indisponible après plusieurs tentatives"}
```



Retry

Réessayer la connexion après un échec



Timeout

Limiter le temps d'attente d'une réponse



Circuit Breaker

Prévenir les appels à un service défaillant

Sécurité dans les Microservices

La sécurité est un aspect crucial dans une architecture de microservices. Chaque service doit être sécurisé individuellement, tout en assurant une communication sécurisée entre les services. Les principales considérations de sécurité incluent l'authentification, l'autorisation, et le chiffrement des données en transit.

L'utilisation de JSON Web Tokens (JWT) pour l'authentification entre services est une pratique courante. Voici un exemple simplifié d'authentification avec JWT en Flask :

```
from flask import Flask, jsonify, request
import jwt

app = Flask(__name__)
app.config['SECRET_KEY'] = 'votre_clé_secrète'

@app.route('/api/login', methods=['POST'])
def login():
    # Vérification des identifiants (simplifié)
    if request.json['username'] == 'admin' and request.json['password'] == 'password':
        token = jwt.encode({'user': request.json['username']}, app.config['SECRET_KEY'], algorithm='HS256')
        return jsonify({'token': token})
    return jsonify({'message': 'Invalid credentials'}), 401
```

1

Authentification

Vérifier l'identité des utilisateurs et des services

2

Autorisation

Contrôler l'accès aux ressources et aux fonctionnalités

3

Chiffrement

Protéger les données sensibles en transit et au repos



Monitoring et Logging

Le monitoring et le logging sont essentiels pour maintenir la santé et la performance d'une architecture de microservices. Ils permettent de détecter rapidement les problèmes, d'analyser les performances et de faciliter le débogage.

Voici un exemple simple d'ajout de logging dans un service Flask :

```
import logging
from flask import Flask, request

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)

@app.route('/api/produits', methods=['GET'])
def obtenir_produits():
    logging.info(f"Requête reçue pour obtenir les produits. IP: {request.remote_addr}")
    # Logique pour récupérer les produits
    return jsonify({"produits": ["Produit1", "Produit2"]})
```

Outil	Fonction
Prometheus	Collecte de métriques
Grafana	Visualisation des données
ELK Stack	Agrégation et analyse des logs

Déploiement et Orchestration

Le déploiement et l'orchestration des microservices sont cruciaux pour gérer efficacement un système distribué. Les conteneurs Docker sont souvent utilisés pour packager les microservices, tandis que Kubernetes est largement adopté pour l'orchestration.

Voici un exemple simplifié de Dockerfile pour un microservice Flask :

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["flask", "run", "--host=0.0.0.0"]
```



Docker
Conteneurisation des microservices pour un déploiement cohérent



Kubernetes
Orchestration pour gérer le déploiement, la mise à l'échelle et la résilience



CI/CD
Intégration et déploiement continus pour une livraison rapide et fiable

Plan du cours :

- 1) Introduction aux APIs
- 2) Gestion de fichiers semi-structurés
- 3) Gestion de bases de données
- 4) Interface utilisateur / API avec Swagger UI

Plan du cours :

- 1) Introduction aux APIs
- 2) Gestion de fichiers semi-structurés
- 3) Gestion de bases de données
- 4) Interface utilisateur / API avec Swagger UI