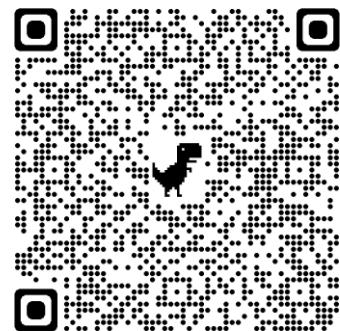




**RÉSUMÉ THÉORIQUE**  
**FILIÈRE DÉVELOPPEMENT DIGITAL – OPTION WEB FULL STACK**  
**M214 – Créer une application cloud native**



**90 heures**



# Equipe de rédaction et de lecture



## Equipe de rédaction :

Mme BOUROUS Imane: Formatrice en développement digital option Web Full Stack

Mme YOUALA Asmae: Formatrice en développement digital option Web Full Stack

## Equipe de lecture :

# SOMMAIRE



## 1. INTRODUIRE LE CLOUD NATIVE

- Définir le cloud
- Définir l'approche cloud native

## 2. CRÉER DES APIS REST SIMPLES EN NODE JS ET EXPRESS JS

- Introduire Express et Node js
  - Créer des APIs REST
- Authentifier une API REST avec JWT

## 3. CRÉER UNE APPLICATION MICROSERVICE

- S'initier aux architectures microservices
  - Créer une application microservices

## 4. MANIPULER LES CONTENEURS

- Appréhender la notion du conteneur
  - Prendre en main Docker

## 5. DÉPLOYER UNE APPLICATION CLOUD NATIVE EN AZURE CLOUD

- Introduire Azure Cloud
- Déployer en Azure App service



## Partie 1

### Introduire le cloud native

Dans cette partie, vous allez :

- Définir le cloud
- Définir l'approche cloud native





## Ce que vous allez apprendre dans ce chapitre :

- Concept du cloud et ses avantages ;
- Exemple des fournisseurs cloud ;
- Différence entre cloud privé, public et hybride ;
- Services du cloud (IAAS, PAAS, SAAS).





# CHAPITRE 1

## Définir le cloud

1. Concept du cloud et ses avantages ;
2. Exemple des fournisseurs cloud ;
3. Différence entre cloud privé, public et hybride ;
4. Services du cloud (IAAS, PAAS, SAAS).

## 1. Définir le cloud

### Concept du cloud et ses avantages

- Le terme « **cloud** » désigne les serveurs accessibles sur Internet, ainsi que les logiciels et bases de données qui fonctionnent sur ces serveurs.
- Les serveurs situés dans le cloud sont hébergés au sein de **datacenters** répartis dans le monde entier.
- L'utilisation du **cloud computing** (informatique cloud) permet aux utilisateurs et aux entreprises de se libérer de la nécessité de gérer des serveurs physiques eux-mêmes ou d'exécuter des applications logicielles sur leurs propres équipements.



## 1. Définir le cloud

### *Concept du cloud et ses avantages*

- Le cloud permet aux utilisateurs d'accéder aux mêmes fichiers et aux mêmes applications à partir de presque n'importe quel appareil, car les processus informatiques et le stockage ont lieu sur des serveurs dans un **datacenter** et non localement sur la machine utilisateur.
- C'est pourquoi vous pouvez vous connecter votre compte Instagram à partir de n'importe quel appareil, avec toutes vos photos, vidéos et l'historique de vos conversations. Il en va de même avec les fournisseurs de messagerie cloud comme Gmail ou Microsoft Office 365 et les fournisseurs de stockage cloud comme Dropbox ou Google Drive.
- Pour les entreprises, le passage au cloud computing supprime certains coûts et frais informatiques : par exemple, les sociétés n'ont plus besoin de mettre à jour et d'entretenir leurs propres serveurs, c'est le fournisseur de cloud qui s'en charge.



## 1. Définir le cloud

### *Concept du cloud et ses avantages*



### **Serveur informatique vs cloud privé : quelle solution de stockage de données choisir pour une entreprise ?**

La question du stockage des données se pose pour toute entreprise. Le volume des données numériques à gérer ne cesse d'augmenter. Optimiser la gestion des documents et le traitement des informations permet aux entreprises de rester concurrentielles.

Concrètement, un serveur informatique relie un poste jouant le rôle de serveur à différents postes utilisateurs (postes clients) et met ces derniers en réseau. Le serveur permet ainsi à chaque client de bénéficier de services divers :

- Le courrier électronique,
- L'accès à Internet,
- Le partage de fichiers,
- Le partage d'imprimantes,
- Le stockage en base de données ,
- La mise à disposition d'applications, etc.

## 1. Définir le cloud

### *Concept du cloud et ses avantages*

Le client se connecte au réseau de l'entreprise et accède à ses documents. Le partage de documents entre les différents membres d'une équipe est également possible mais uniquement sur les postes installés en interne au sein de l'entreprise.

### **Les limites du serveur informatique:**

#### => La sécurité des données en question

L'utilisation d'un support de stockage expose les entreprises à d'autres risques :

- pannes matérielles pouvant rendre les systèmes de gestion inopérants ;
- infestation des données (introduction d'un malware dans les systèmes informatiques) ou piratage des données.
- Une capacité de stockage limitée
- Des coûts élevés pour l'entreprise



## 1. Définir le cloud

### *Concept du cloud et ses avantages*

le cloud computing doit posséder 4 caractéristiques essentielles :

**Le service doit être en libre-service à la demande**

**Le service doit être mesurable (mesure et affichage de paramètres de consommation).**

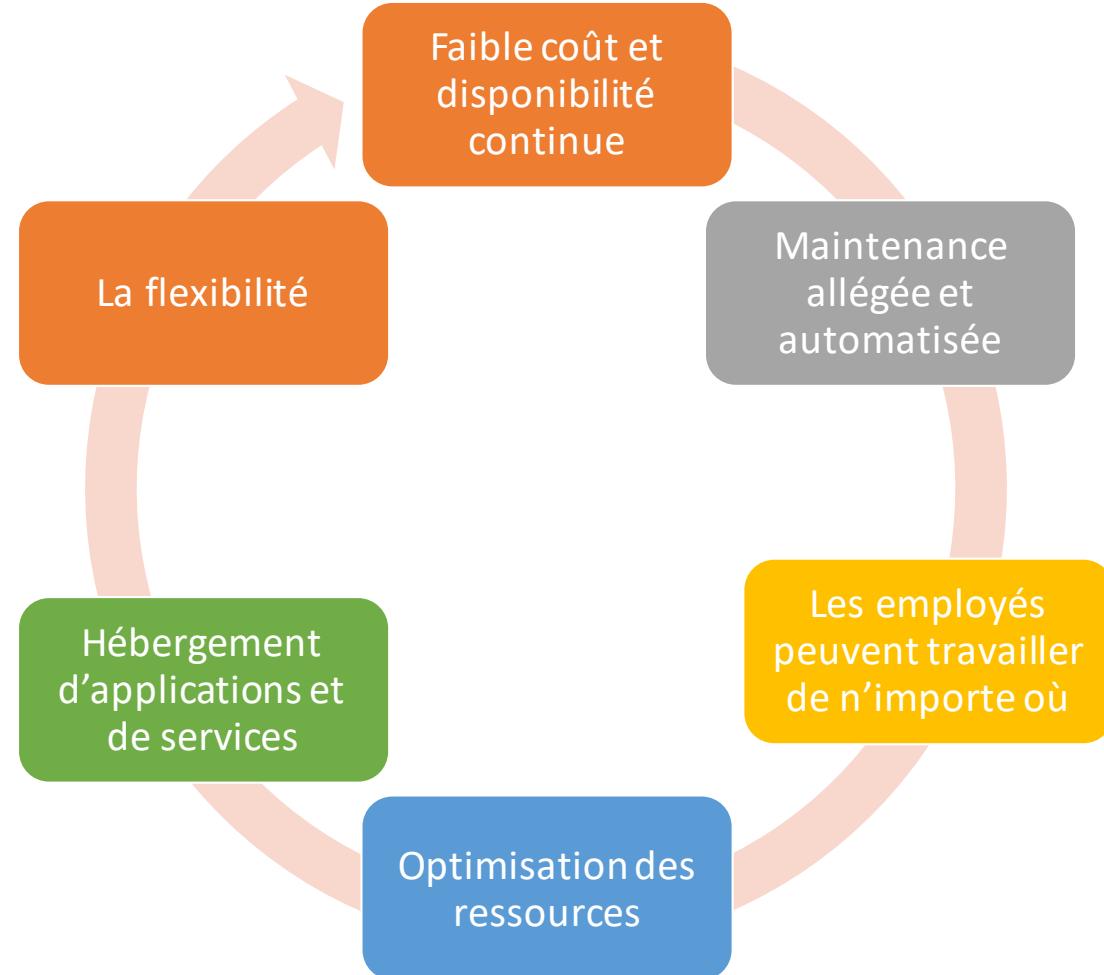
**Il doit y avoir une mutualisation des ressources**

**Il doit être rapidement élastique (adaptation rapide à une variation du besoin)**

## 1. Définir le cloud

### Concept du cloud et ses avantages

- **Les avantages du Cloud**





# CHAPITRE 1

## Définir le cloud

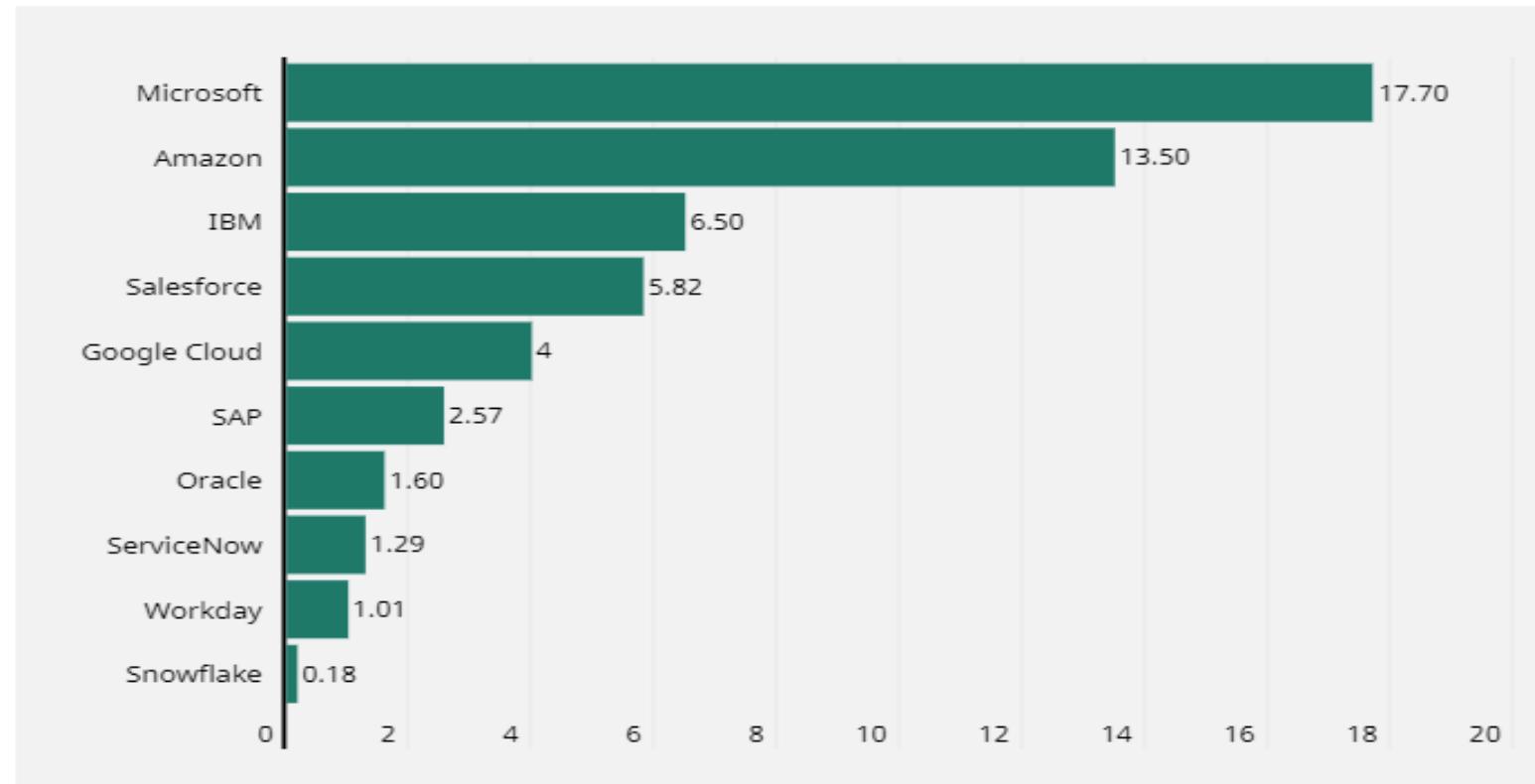
1. Concept du cloud et ses avantages ;
2. **Exemple des fournisseurs cloud ;**
3. Différence entre cloud privé, public et hybride ;
4. Services du cloud (IAAS, PAAS, SAAS).

## 1. Définir le cloud

### Exemple des fournisseurs cloud

- Les 10 premiers fournisseurs mondiaux de cloud en termes de revenus totaux pour le trimestre fiscal se terminant le 31 mars 2021 (en milliards de dollars américains)

Source: Statista, cloudwars.co





# CHAPITRE 1

## Définir le cloud

1. Concept du cloud et ses avantages ;
2. Exemple des fournisseurs cloud ;
3. **Différence entre cloud privé, public et hybride ;**
4. Services du cloud (IAAS, PAAS, SAAS).

## 1. Définir le cloud

### Différence entre cloud privé, public et hybride

#### Cloud public



- Les clouds **publics** sont généralement des environnements cloud créés à partir d'une infrastructure informatique qui n'appartient pas à l'utilisateur final.
- Alibaba Cloud, Microsoft Azure, Google Cloud, Amazon Web Services (AWS) et IBM Cloud sont les principaux fournisseurs de cloud public.
- Les clouds **publics** étaient habituellement exécutés hors site, mais les fournisseurs de cloud public proposent désormais des services cloud dans les datacenters de leurs clients, ce qui rend les notions d'emplacement et de propriété obsolètes.

## 1. Définir le cloud

### *Difference entre cloud privé, public et hybride*

#### Cloud public



- Les clouds **privés** sont généralement définis comme des environnements cloud spécifiques à un utilisateur final ou à un groupe, et sont habituellement exécutés derrière le pare-feu de l'utilisateur ou du groupe.
- Tous les clouds deviennent des clouds **privés** lorsque l'infrastructure informatique sous-jacente est spécifique à **un client unique**, avec un accès entièrement **isolé**.

## 1. Définir le cloud

### *Difference entre cloud privé, public et hybride*



#### Cloud public

Toutefois, les clouds **privés** ne reposent désormais plus forcément sur une infrastructure informatique sur site. Aujourd'hui, les entreprises créent des clouds privés dans des **datacenters hors site** et loués à des fournisseurs, ce qui rend les règles relatives à l'emplacement et à la propriété obsolètes.

Cette tendance a fait naître différents sous-types de clouds privés, notamment :

- ✓ Clouds privés gérés: Ce type de cloud est créé et utilisé par les clients, tandis qu'il est déployé, configuré et géré par un fournisseur tiers.
- ✓ Clouds dédiés: Il s'agit d'un cloud au sein d'un autre cloud. Vous pouvez déployer un cloud spécialisé dans un cloud public.

## 1. Définir le cloud

### Différence entre cloud privé, public et hybride

#### Cloud hybride

Un **cloud hybride** fonctionne comme un environnement informatique unique créé à partir de plusieurs environnements connectés via des réseaux locaux (LAN), des réseaux étendus (WAN), des réseaux privés virtuels (VPN) et/ou des API.

Les caractéristiques des clouds hybrides sont complexes et les exigences associées peuvent varier selon l'utilisateur qui les définit. Par exemple, un cloud hybride peut inclure :

- ✓ Au moins un cloud privé et au moins un cloud public
- ✓ Au moins deux clouds privés
- ✓ Au moins deux clouds publics
- ✓ Un environnement virtuel connecté à au moins un cloud privé ou public



# CHAPITRE 1

## Définir le cloud

1. Concept du cloud et ses avantages ;
2. Exemple des fournisseurs cloud ;
3. Différence entre cloud privé, public et hybride ;
4. **Services du cloud (IAAS, PAAS, SAAS).**

# 1. Définir le cloud

## Services du cloud (IAAS, PAAS, SAAS)

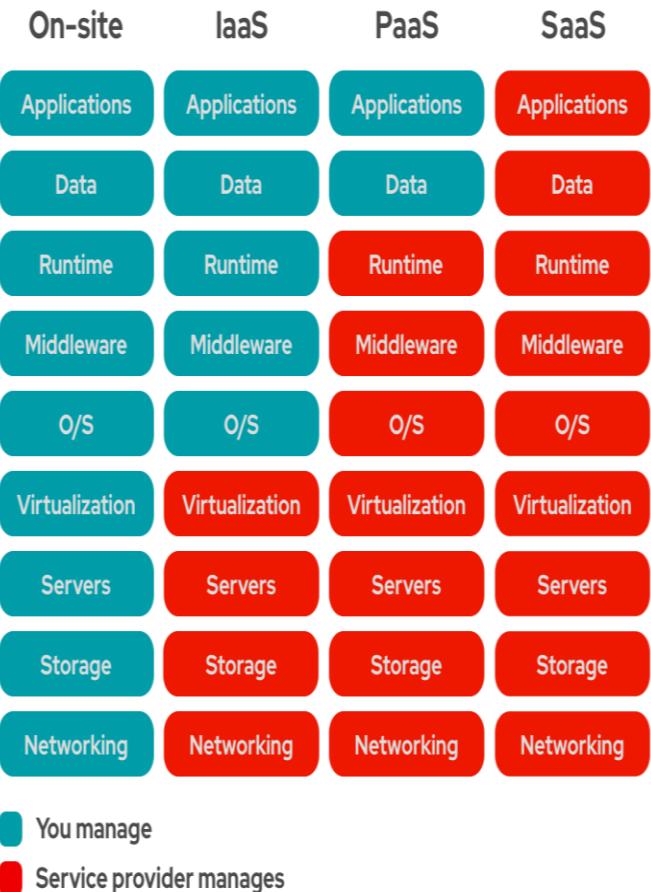
### As-a-Service : définition

L'expression « **aas** » ou « as-a-Service » signifie généralement qu'un tiers se charge de vous fournir un service de cloud computing, afin que vous puissiez vous concentrer sur des aspects plus importants, tels que votre code et les relations avec vos clients.

Chaque type de cloud computing allège la gestion de votre infrastructure sur site.

Il existe trois principaux types de cloud computing « **as-a-Service** », chacun offrant un certain degré de gestion :

- **IaaS** (Infrastructure-as-a-Service)
- **PaaS** (Platform-as-a-Service)
- **SaaS** (Software-as-a-Service).



## 1. Définir le cloud

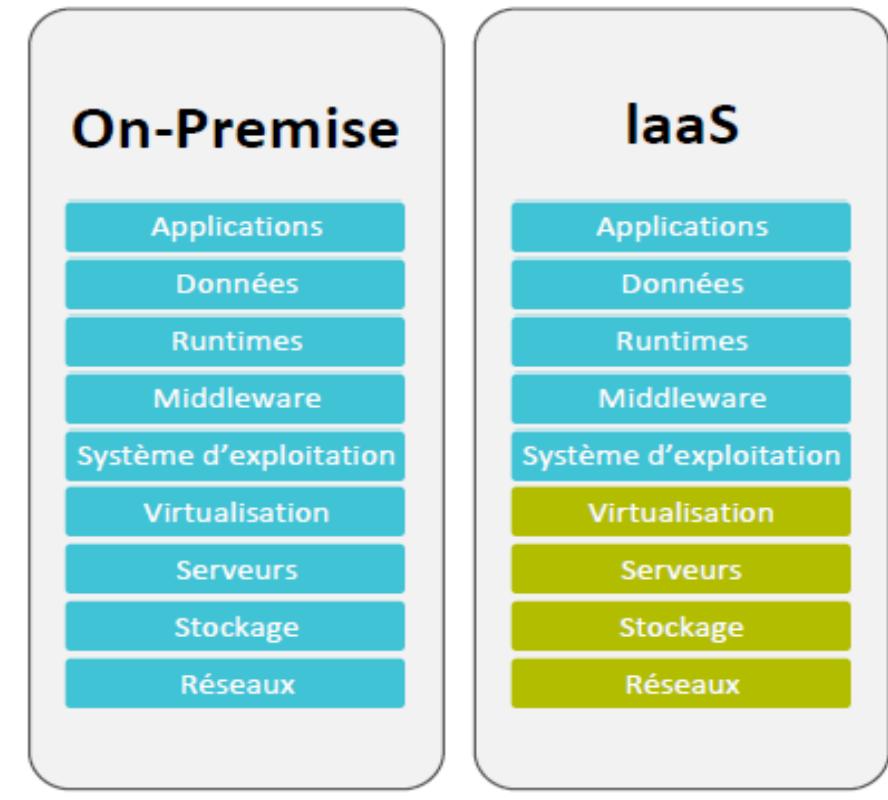
### *Services du cloud (IAAS, PAAS, SAAS)*

#### IaaS : Infrastructure as a Service

Pour ce type de service le fournisseur de solution fournit les fonctions de virtualisation le système de stockage les réseaux et les serveurs et vous y donne accès en fonction de vos besoins;

Ainsi, l'utilisateur ne contrôle pas l'infrastructure Cloud sous jacente et il n'a pas à s'inquiéter des mises à jour physiques ou de la maintenance de ces composants;

Par contre et en tant qu'utilisateur, vous êtes responsable du **système d'exploitation** ainsi que **des données applications**, solutions de middleware et environnements d'exécution.



 Vous gérez

 Gérer par le fournisseur de services

# 1. Définir le cloud

## Services du cloud (IAAS, PAAS, SAAS)

### IaaS : Infrastructure as a Service

L'IaaS est le modèle Cloud « as a Service » le plus flexible et libre, il apporte aux utilisateurs tous les avantages des ressources informatiques sur site, sans les actions et frais de gestion de l'infrastructure

En effet, il facilite la mise à l'échelle, la mise à niveau et permet d'ajouter des ressources, par exemple le stockage dans le Cloud

#### Exemples

Fournisseurs	AWS	Google Cloud	Azure
IaaS Services	Elastic ComputeCloud (EC2)	Compute Engine	Virtual Machine

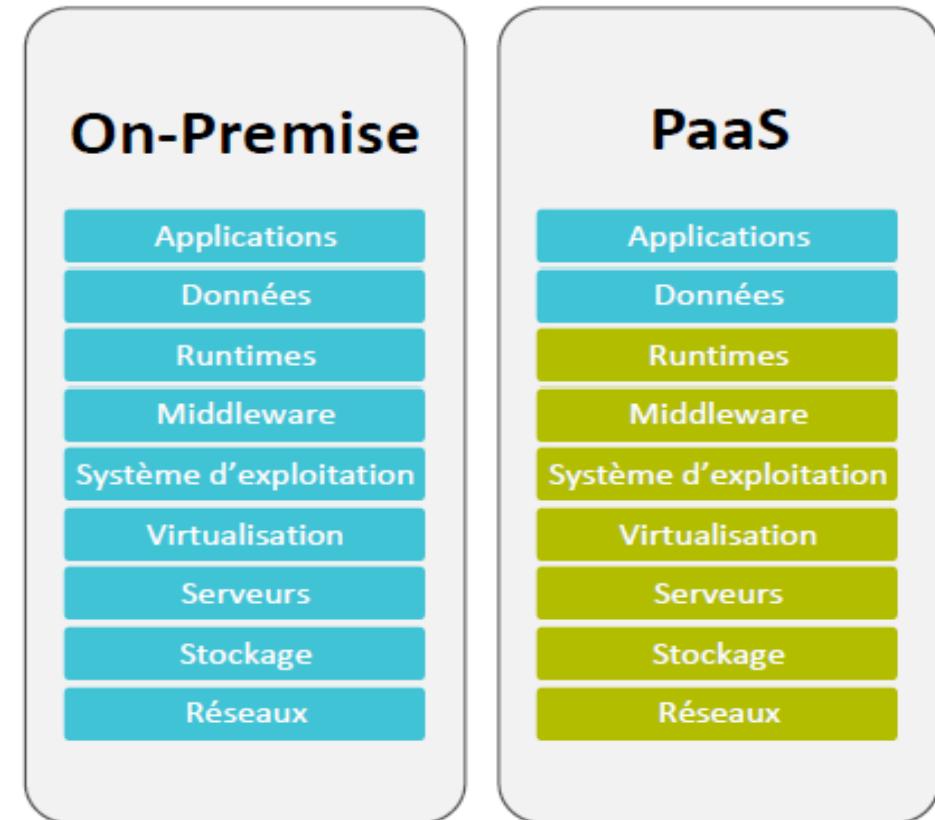
## 1. Définir le cloud

### *Services du cloud (IAAS, PAAS, SAAS)*

#### PaaS : Platform-as-a-Service

Le type de service **PaaS** est semblable à du IaaS, sauf que votre fournisseur de services Cloud fournit également le système d'exploitation et les environnements d'exécutions.

- Ainsi, l'utilisateur ne contrôle pas l'infrastructure Cloud sous-jacente et il n'a pas à s'inquiéter des mises à jour physiques ou de la maintenance de ces composants y compris le réseaux, les serveurs, les systèmes d'exploitations ou de stockage.
- Par contre et en tant qu'utilisateur, vous avez le contrôle pour le déploiement et configuration d'applications créées à l'aide de langages de programmation, de bibliothèques, de services et d'outils pris en charge par le fournisseur.



Vous gérez

Gérer par le fournisseur de services

## 1. Définir le cloud

### Services du cloud (IAAS, PAAS, SAAS)

#### PaaS : Platform-as-a-Service

- Idéalement destiné aux développeurs et aux programmeurs, le PaaS fournit une plateforme simple et évolutive permettant aux utilisateurs d'exécuter et gérer leurs propres applications, sans avoir à créer ni entretenir l'infrastructure ou la plateforme généralement associée au processus.

#### Exemples

Fournisseurs	AWS	aws	Google Cloud	Google Cloud logo	Azure	Azure
PaaS services	AWS Elastic Beanstalk		Google App Engine		Azure App Service	Azure function App

*Un service de gestion base de données géré par le fournisseur et accessible via le Cloud est considéré comme du PaaS. Exemple : Azure SQL DB, Azure Cosmos DB ...*

## 1. Définir le cloud

### **Services du cloud (IAAS, PAAS, SAAS)**

#### **SaaS Software as a Service**

Le SaaS (ou services d'applications Cloud, est le type le plus complet qui utilise le plus des services sur le marché du Cloud

Pour ce type de service le fournisseur fournit et gère une application complète accessible par les utilisateurs via un navigateur Web ou un client lourd

Ainsi, l'utilisateur ne contrôle pas la plateforme Cloud sous jacente et il n'a pas à s'inquiéter des mises à jour logicielles ou l'application des correctifs et les autres tâches de maintenance logicielle

#### **On-Premise**

- Applications
- Données
- Runtimes
- Middleware
- Système d'exploitation
- Virtualisation
- Serveurs
- Stockage
- Réseaux

#### **SaaS**

- Applications
- Données
- Runtimes
- Middleware
- Système d'exploitation
- Virtualisation
- Serveurs
- Stockage
- Réseaux



**Vous gérez**

**Gérer par le fournisseur de services**

# 1. Définir le cloud

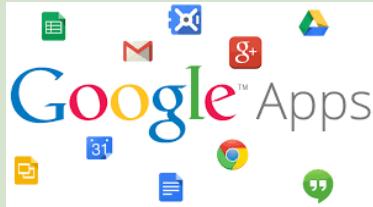
## Services du cloud (IAAS, PAAS, SAAS)

### SaaS Software as a Service

Le SaaS constitue une option intéressante pour les PME qui n'ont pas les ressources humaines pour gérer l'installation et le suivi de l'installation des mises à jour de sécurité et logiciels.

Par ailleurs, il est à noter que le modèle SaaS réduit le niveau de contrôle et peut nuire à la sécurité et aux performances => Il convient donc de choisir soigneusement votre fournisseur Cloud

### Exemples

Fournisseurs	AWS 	Google Cloud 	Azure 
SaaS services	Zoom 	Google Apps 	Microsoft Office 365 



## CHAPITRE 2

### Définir l'approche cloud native

**Ce que vous allez apprendre dans ce chapitre :**

- Définition de l'approche cloud native
- Avantages
- Vue générale sur les caractéristiques du cloud natif :
- ✓ Automatisation des processus du développement et de déploiement,
- ✓ Microservices et conteneurs.





## CHAPITRE 2

### Définir l'approche cloud native

1. **Définition;**
2. Avantages ;
3. Vue générale sur les caractéristiques du cloud natif :

## 2. Définir l'approche cloud native

### Définition

**Cloud Native** : le Cloud Native décrit une approche de développement logiciel dans laquelle les applications sont dès le début conçues pour une utilisation sur le Cloud.

Il en résulte des applications Cloud Native (NCA) capables de pleinement exploiter les atouts de l'architecture du **Cloud Computing**.

Cette approche se concentre sur **le développement d'applications sous la forme de microservices individuels**, qui ne sont pas exécutés « On-Premises » (localement), mais sur **des plateformes agiles basées sur des conteneurs**.

Cette approche accélère le développement de logiciels et favorise la création d'applications **résilientes et évolutives**.



## 2. Définir l'approche cloud native

### Définition

#### Fonctionnement

L'approche Cloud Native repose sur **quatre piliers** qui sont liés et interdépendants.

- Du côté de **technique**, on trouve les **microservices** et les technologies de **conteneurs** développées spécialement pour l'environnement Cloud qui constituent des éléments fondamentaux du concept Cloud Native. Les différents microservices remplissent une fonction précise et peuvent être rassemblés dans un conteneur avec tout ce qui est nécessaire à leur exécution. Ces conteneurs sont portables et offrent aux équipes de développement un haut degré de flexibilité, par exemple lorsqu'il s'agit de tester de nouveaux services.
- Du côté de la **stratégie**, les **processus de développement** et la **Continuous Delivery** sont bien établis. Lors de la conception d'une architecture Cloud Native efficace, les équipes de développeurs (Developers = Dev), mais aussi l'entreprise (Operations = Ops) sont directement impliquées. Dans le cadre d'un échange constant, l'équipe de développeurs ajoute à un microservice certaines fonctionnalités livrées automatiquement par des processus de **Continuous-Delivery**.



## CHAPITRE 2

### Définir l'approche cloud native

1. Définition ;
2. **Avantages ;**
3. Vue générale sur les caractéristiques du cloud natif :

## 2. Définir l'approche cloud native

### Avantages

#### Avantages

- ✓ **Flexibilité:** Comme tous les services sont exécutés indépendamment de leur environnement les développeurs disposent d'une **grande liberté**. **Les modifications apportées au code n'ont pas d'impact sur le logiciel dans son ensemble.** Le déploiement de nouvelles versions du logiciel présente donc un risque plus faible.
- ✓ **L'évolutivité** des applications à proprement parler, qui permet aux entreprises de ne pas devoir procéder à **une mise à niveau coûteuse du matériel** en cas d'augmentation des exigences pour un service.
- ✓ Le haut **niveau d'automatisation** réduit par ailleurs à un minimum les erreurs humaines de configuration et d'utilisation.

## 2. Définir l'approche cloud native

### Avantages

#### Avantages

- ✓ Voici quelques entreprises qui ont implémenté des techniques natives Cloud et qui ont obtenu, par conséquence, la vitesse, l'agilité et la scalabilité.
- ✓ Netflix, Uber et WeChat exposent des systèmes natifs Cloud qui se composent de nombreux services indépendants. Ce style architectural leur permet de répondre rapidement aux conditions du marché. Elles mettent instantanément à jour de petites zones d'une application complexe en service, sans redéploiement complet. Elles mettent à l'échelle individuellement les services en fonction des besoins.

Entreprise	Expérience
<b>NETFLIX</b>	Dispose de plus de 600 services en production. Effectue des déploiements 100 fois par jour.
<b>Uber</b>	Dispose de plus de 1 000 services en production. Effectue des déploiements plusieurs milliers de fois par semaine.
 <b>WeChat</b>	Dispose de plus de 3 000 services en production. Effectue des déploiements 1 000 fois par jour.



## CHAPITRE 2

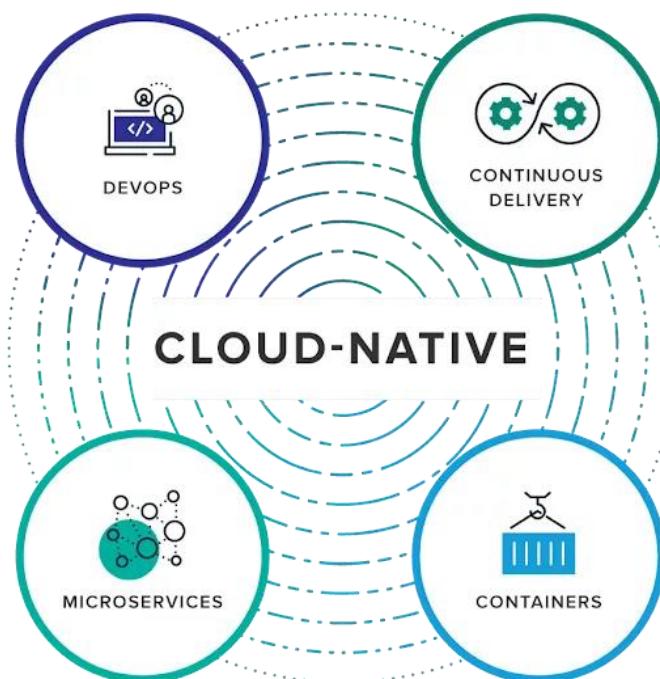
### Définir l'approche cloud native

1. Définition ;
2. Avantages ;
3. **Vue générale sur les caractéristiques du cloud natif :**

## 2. Définir l'approche cloud native

### *Vue générale sur les caractéristiques du cloud natif*

L'approche Cloud Native, se caractérise par l'utilisation d'architectures en **microservices**, de la technologie de **conteneurs**, de **livraisons en continu**, de **pipelines** de développement et d'infrastructure exprimés sous forme de code (Infrastructure As a Code), une pratique importante de la culture **DevOps**.



## 2. Définir l'approche cloud native

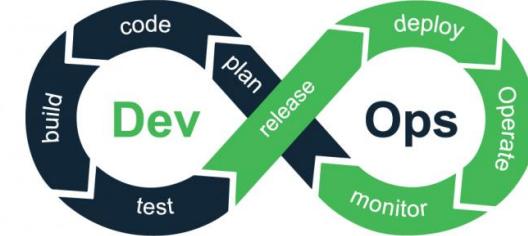
### *Vue générale sur les caractéristiques du cloud natif*

#### Automatisation des processus du développement et de déploiement:

Comme l'approche DevOps, le **Cloud Native** cherche à rassembler les équipes Dev et Ops autour d'un objectif commun long terme : celui de la création de valeur business par les applications.

L'approche DevOps permet de converger vers une **approche Cloud Native** avec l'automatisation des processus et des technologies entre les équipes, de façon à intégrer plus rapidement les innovations dans les cycles de développement et de déploiement d'une **application Cloud Native**.

En parallèle du **Cloud Native**, l'adoption des méthodes Agiles va permettre d'intégrer les équipes métier dans cette collaboration avec les équipes techniques et de développement. L'idée est de collaborer pour délivrer une itération en améliorant le produit à chaque livraison de façon continue.



## 2. Définir l'approche cloud native

### *Vue générale sur les caractéristiques du cloud natif*

#### Les microservices

Les **microservices** désignent à la fois une architecture et une approche de développement logiciel qui consiste à décomposer les applications en éléments les plus simples, indépendants les uns des autres. Contrairement à une approche monolithique classique, selon laquelle tous les composants forment une entité indissociable, les microservices fonctionnent en synergie pour accomplir les mêmes tâches, tout en étant séparés.

Pour communiquer entre eux, les **microservices** d'une application utilisent le modèle de communication requête-réponse. L'implémentation typique utilise des appels API REST basés sur le protocole HTTP. Les procédures internes (appels de fonctions) facilitent la communication entre les composants de l'application.

les microservices sont beaucoup plus faciles à créer,  
tester, déployer et mettre à jour

## 2. Définir l'approche cloud native

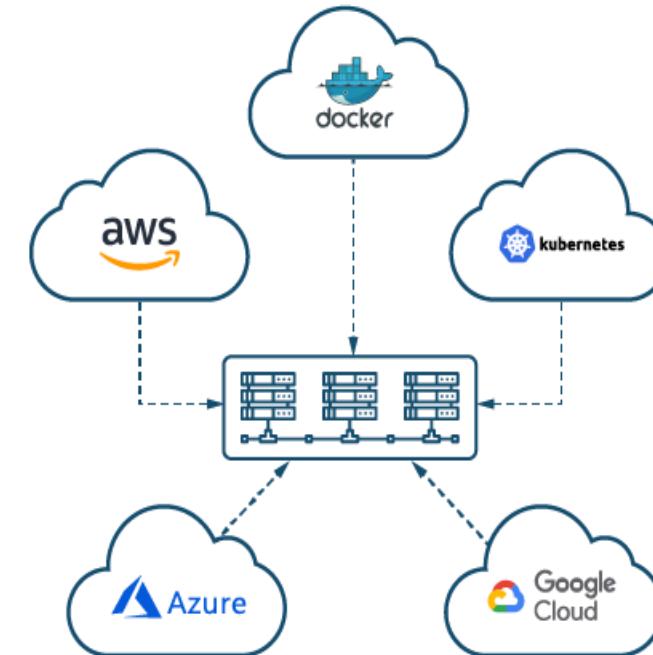
### Vue générale sur les caractéristiques du cloud natif

#### Les Conteneurs

Tout comme le secteur du transport utilise des conteneurs pour isoler les différentes marchandises à transporter à bord des navires, des trains, des camions et des avions, le développement logiciel a de plus en plus recours au concept de **conteneurisation**.

Un package logiciel unique, appelé « **conteneur** », regroupe le code d'une application avec les fichiers de configuration, les bibliothèques et les dépendances requises pour que l'application puisse s'exécuter.

Ceci permet aux développeurs et aux professionnels de l'informatique de déployer les applications de façon **transparente dans tous les environnements**.





## Partie 2

# CRÉER DES APIS REST SIMPLES EN NODE JS ET EXPRESS JS

Dans cette partie, vous allez :

- Introduire Express et Node js
- Créer des APIs REST
- Authentifier et autoriser une API REST avec JWT





## CHAPITRE 1

### Introduire Express et Node js

Ce que vous allez apprendre dans ce chapitre :

- Rappel du concept des APIs REST ;
- Rappel des méthodes du protocole http ;
- Définition de l'ecosystème Node JS ;
- Configuration de l'environnement de développement;
- L'essentiel du Node js



# CHAPITRE 1

## Introduire Express et Node js

- 1. Rappel du concept des APIs REST ;**
2. Rappel des méthodes du protocole http ;
3. Définition de l'ecosystème Node JS ;
4. Configuration de l'environnement de développement;
5. L'essentiel du Node js

Une API (**interface de programmation d'application**) est un ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications.

l'API est **l'intermédiaire** permettant à deux systèmes informatiques totalement indépendants d'interagir entre eux, de manière automatique, sans intervention humaine.

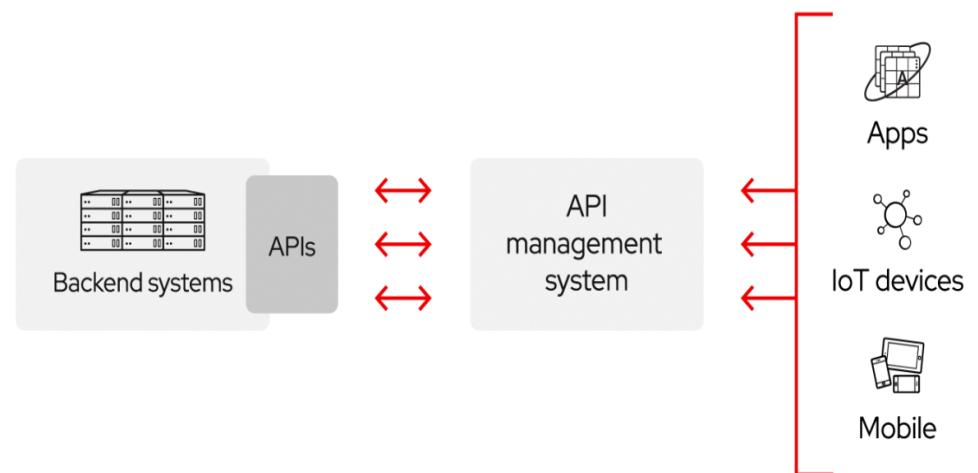
Elle est parfois considérée comme un **contrat** entre un **fournisseur** d'informations et un **utilisateur** d'informations, qui permet de définir le contenu demandé au consommateur (l'appel) et le contenu demandé au producteur (la réponse).

Par exemple, l'API conçue pour un service de météo peut demander à l'utilisateur de fournir un code postal et au producteur de renvoyer une réponse en deux parties : la première concernant la température maximale et la seconde la température minimale.



### Avantages des APIs

- ✓ Elle permet de pouvoir interagir avec un système sans se soucier de sa complexité et de son fonctionnement. ...
- ✓ Une API est souvent spécialisée dans un domaine et sur un use case particulier ce qui simplifie son utilisation, sa compréhension et sa sécurisation.
- ✓ Les API constituent un moyen simplifié de connecter votre propre infrastructure au travers du développement d'applications cloud-native.
- ✓ Elles vous permettent également de partager vos données avec vos clients et d'autres utilisateurs externes.
- ✓ Les API publiques offrent une valeur métier unique, car elles peuvent simplifier et développer vos relations avec vos partenaires, et éventuellement monétiser vos données (l'API Google Maps en est un parfait exemple)



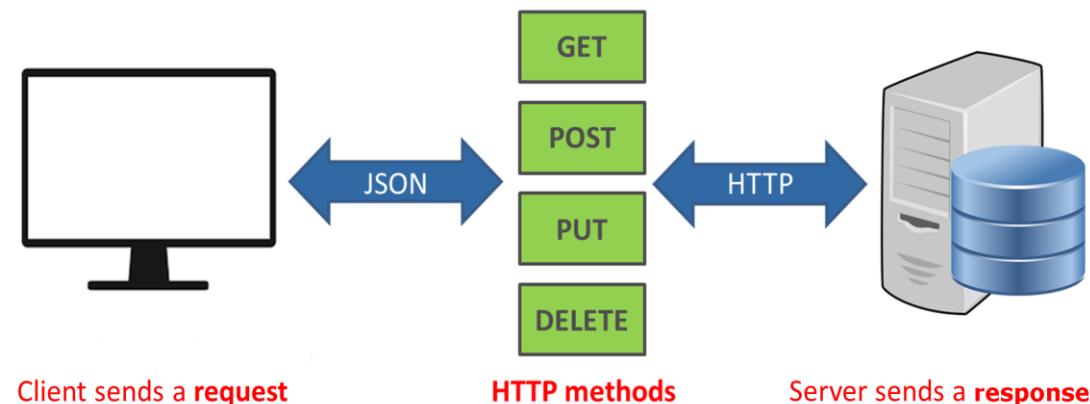
### I'API REST ?

Roy Fielding a défini REST comme un style architectural et une méthodologie fréquemment utilisés dans le développement de services Internet, tels que les systèmes hypermédias distribués.

La forme complète de l'API REST est l'interface de programmation d'applications de transfert d'état représentationnelle, plus communément appelée service Web API REST.

Par exemple, lorsqu'un développeur demande à l'API Twitter de récupérer l'objet d'un utilisateur (une ressource), l'API renvoie l'état de cet utilisateur, son nom, ses abonnés et les publications partagées sur Twitter. Cela est possible grâce aux projets d'intégration d'API.

Cette représentation d'état peut être au format JSON, XML ou HTML.



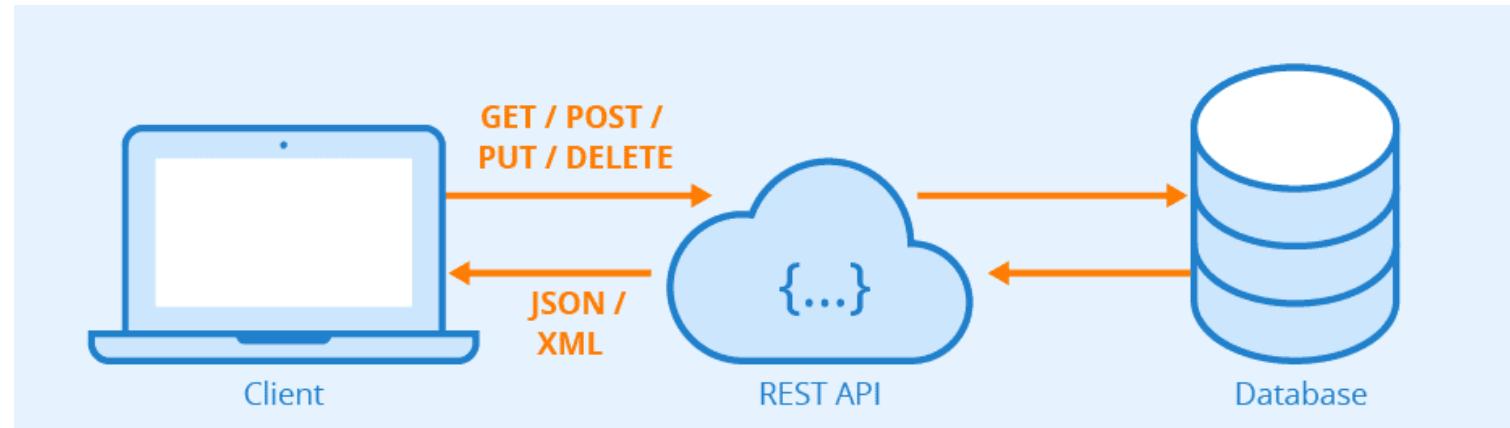
### Comment fonctionne une API REST?

REST détermine la structure d'un API. Les développeurs s'obligent à un ensemble de règles spécifiques lors de la conception d'une API. Par exemple, une loi stipule qu'un lien vers une URL doit renvoyer certaines informations.

Chaque URL est connue sous le nom de demande (request), tandis que les données renvoyées sont appelées réponse (response).

L'API REST décompose une transaction pour générer une séquence de petits composants. Chaque composant aborde un aspect fondamental spécifique d'une transaction. Cette modularité en fait une approche de développement flexible.

Une API REST exploite les méthodes HTTP décrites par le Protocole RFC 2616





# CHAPITRE 1

## Introduire Express et Node js

1. Rappel du concept des APIs REST ;
2. **Rappel des méthodes du protocole http ;**
3. Définition de l'ecosystème Node JS ;
4. Configuration de l'environnement de développement;
5. L'essentiel du Node js

**HTTP (Hypertext Transfer Protocol)** est créé pour fournir la communication entre les clients et le serveur.

Il fonctionne en tant qu'une requête et une réponse.

Il existe deux méthodes HTTP principalement utilisées: GET et POST:

#### La méthode GET

La méthode GET de HTTP demande des données d'une source spécifiée. Les demandes GET peuvent être mises en cache et rester dans l'historique du navigateur. Il peut également être marqué.

Il ne doit jamais être utilisé lorsque vous travaillez sur des données sensibles. Les requêtes GET ont des restrictions de longueur et ne doivent être utilisées que pour obtenir des données.

#### La méthode POST

La méthode POST envoie les données à traiter à une source spécifiée. Contrairement à la méthode GET, les requêtes POST ne sont jamais paramétrées, elles ne restent pas dans l'historique du navigateur et nous ne pouvons pas les mettre en signet. De plus, les requêtes POST n'ont aucune restriction de longueur de données.



### Comparaison des méthodes GET et POST

	GET	POST
<b>Peut être marqué</b>	Oui	Non
<b>Peut être mise en cache</b>	Oui	Non
<b>Historique</b>	Reste dans l'historique de navigateur.	Ne reste pas dans l'historique de navigateur.
<b>Restrictions de type de données</b>	Seulement les caractères ASCII sont autorisés.	N'a aucune restriction. Les données binaires sont également autorisées.
<b>Sécurité</b>	Il est moins sécurisé que le POST car les données envoyées font partie de l'URL.	Le POST est un peu plus sûr que GET car il ne reste pas dans l'historique du navigateur ni dans les journaux du serveur Web.
<b>Visibilité</b>	Les données sont visibles à tous dans l'URL.	N'affiche pas les données dans l'URL.

Outre les méthodes GET et POST, il existe d'autres méthodes.

Méthode	Descriptions
<b>HEAD</b>	Il en va de même avec la méthode GET mais elle ne renvoie que des lecteurs HTTP, pas de corps de document.
<b>PUT</b>	Il télécharge la représentation de l'URI spécifié.
<b>DELETE</b>	Il supprime la ressource spécifiée.
<b>OPTIONS</b>	Il retourne les méthodes HTTP supportées par le serveur.
<b>CONNECT</b>	Il convertit la connexion de demande en tunnel TCP / IP transparent.



# CHAPITRE 1

## Introduire Express et Node js

1. Rappel du concept des APIs REST ;
2. Rappel des méthodes du protocole http ;
- 3. Définition de l'ecosystème Node JS ;**
4. Configuration de l'environnement de développement;
5. L'essentiel du Node js

## Introduire Express et Node js

### Définition de l'écosystème Node JS



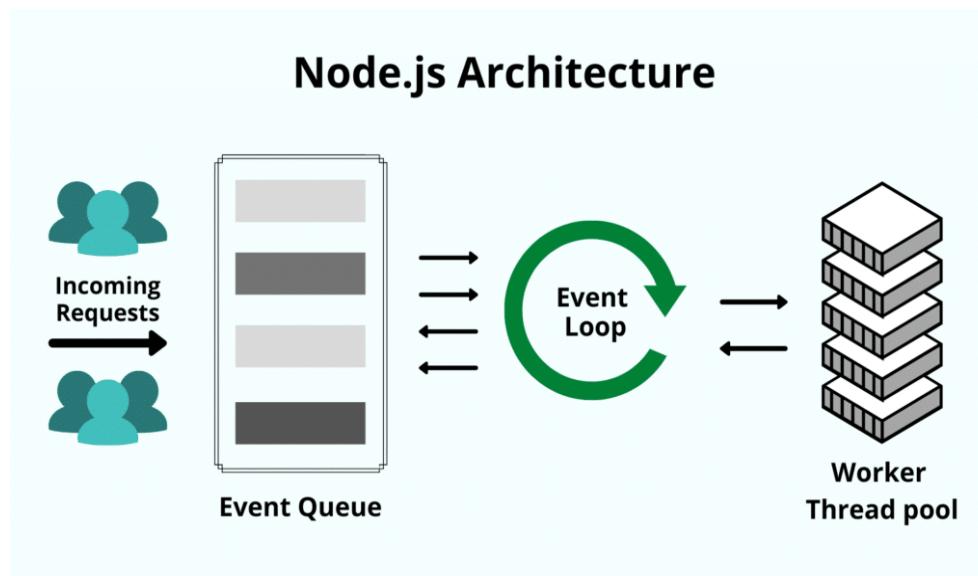
- ✓ **Node.js** est un environnement pour développer et déployer des applications web à base du Javascript.
- ✓ Plusieurs frameworks Javascript permettent de développer la partie frontale tel que Angular, VueJS , React
- ✓ **Node.js** est un environnement d'exécution single-thread, open-source et multi-plateforme permettant de créer des applications rapides et évolutives côté serveur et en réseau.
- ✓ Il fonctionne avec le moteur d'exécution JavaScript V8 et utilise une architecture d'E / S non bloquante et pilotée par les événements, ce qui le rend efficace et adapté aux applications en temps réel.



## Introduire Express et Node js

### Définition de l'écosystème Node JS

- ✓ Node.js utilise l'architecture « Single Threaded Event Loop » pour gérer plusieurs clients en même temps.
- ✓ Pour comprendre en quoi cela est différent des autres runtimes, nous devons comprendre comment les clients concurrents multi-threads sont gérés dans des langages comme Java.
- ✓ Dans un modèle requête-réponse multi-thread, plusieurs clients envoient une requête, et le serveur traite chacune d'entre elles avant de renvoyer la réponse. Cependant, plusieurs threads sont utilisés pour traiter les appels simultanés. Ces threads sont définis dans un pool de threads, et chaque fois qu'une requête arrive, un thread individuel est affecté à son traitement.



### Caractéristiques de Node.js

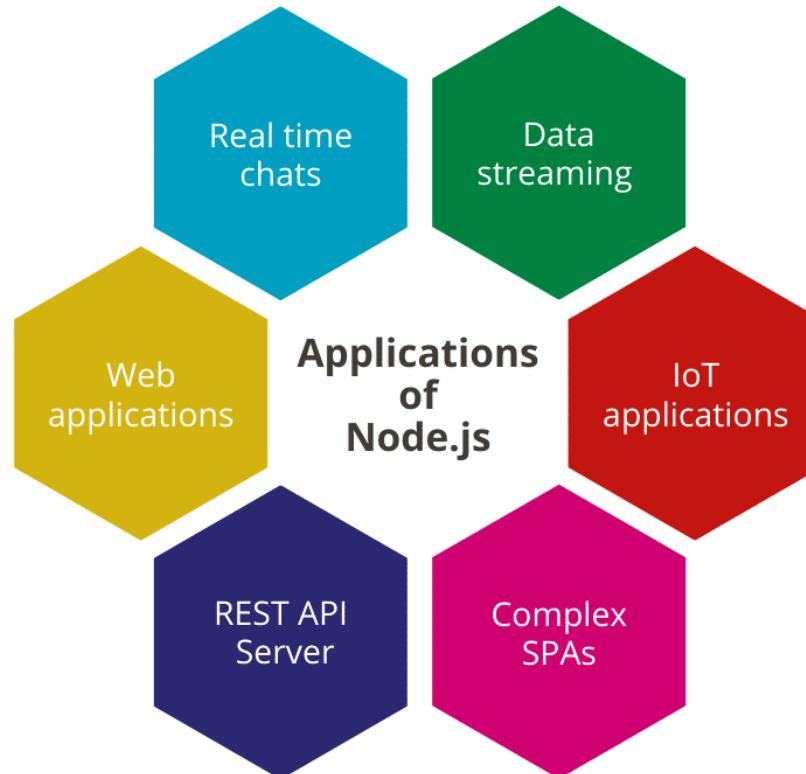
Node.js a connu une croissance rapide au cours des dernières années. Cela est dû à la vaste liste de fonctionnalités qu'il offre :



- ✓ **Facile** : Node.js est assez facile à prendre en main.
- ✓ **Évolutif** – Il offre une grande évolutivité aux applications. Node.js, étant single-thread, est capable de gérer un grand nombre de connexions simultanées avec un débit élevé
- ✓ **Vitesse** – L'exécution non bloquante des threads rend Node.js encore plus rapide et plus efficace.
- ✓ **Paquets** – Un vaste ensemble de paquets Node.js open source est disponible et peut simplifier votre travail. Aujourd'hui, il y a plus d'un million de paquets dans l'écosystème NPM.
- ✓ **Backend solide** – Node.js est écrit en C et C++, ce qui le rend rapide et ajoute des fonctionnalités comme le support réseau.
- ✓ **Multi-plateforme** – La prise en charge multi-plateforme vous permet de créer des sites web SaaS, des applications de bureau et même des applications mobiles, le tout en utilisant Node.js.
- ✓ **Maintenable** – Node.js est un choix facile pour les développeurs, car le frontend et le backend peuvent être gérés avec JavaScript comme un seul langage

Quelques entreprises des plus populaires qui utilisent Node.js aujourd'hui : Twitter, Spotify, eBay, LinkedIn

### Applications de Node.js



## Introduire Express et Node js

### Définition de l'écosystème Node JS



**Express** est un framework Node pour développer la partie Backend

Ainsi il est dorénavant possible de développer des applications fullstackJS

Dans ce module, on va développer des API Rest via Express, Dans un premier temps on va utiliser des données dans des fichiers JSON, ensuite on va traiter le cas MongoDB et MySQL.





# CHAPITRE 1

## Introduire Express et Node js

1. Rappel du concept des APIs REST ;
2. Rappel des méthodes du protocole http ;
3. Définition de l'ecosystème Node JS ;
- 4. Configuration de l'environnement de développement;**
5. L'essentiel du Node js

# Introduire Express et Node js

## Configuration de l'environnement de développement



### Installation de Node.js

Tout d'abord, nous devons télécharger le fichier Windows Installer (.msi) depuis le site officiel de Node.js.

Ce fichier d'installation MSI contient une collection de fichiers d'installation essentiels pour installer, mettre à jour ou modifier la version existante de Node.js.

Le programme d'installation contient également le gestionnaire de paquets Node.js (npm). Cela signifie que vous n'avez pas besoin d'installer npm séparément.

Visiter le site officiel : <https://nodejs.org/en/>

Node.js® is an open-source, cross-platform JavaScript runtime environment.

Node.js assessment of OpenSSL 3.0.7 security advisory

Download for Windows (x64)

18.12.1 LTS

Recommended For Most Users

19.3.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)    [Other Downloads](#) | [Changelog](#) | [API Docs](#)

For information about supported releases, see the [release schedule](#).

# Introduire Express et Node js

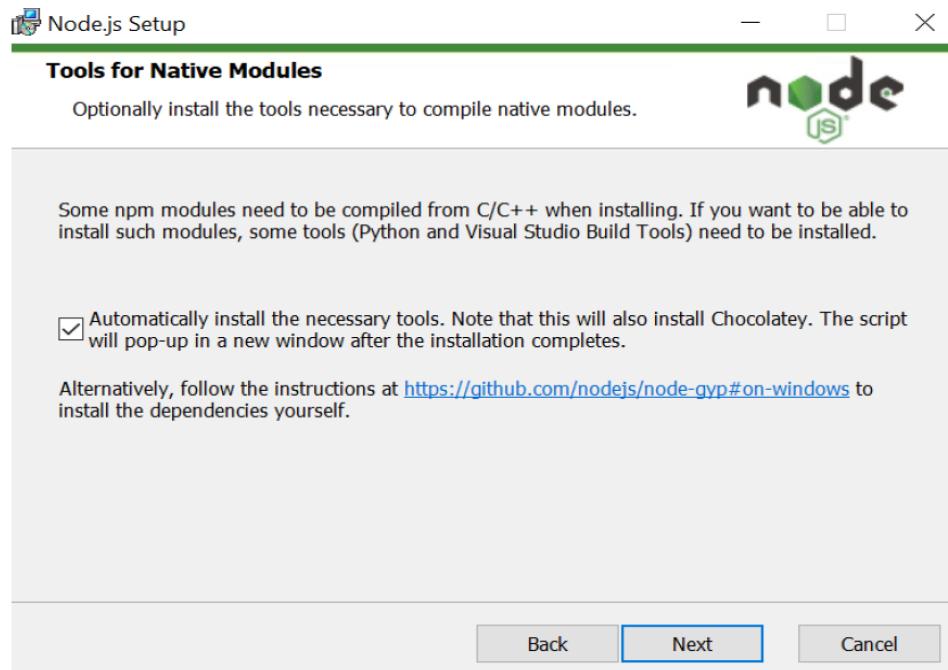
## Configuration de l'environnement de développement



### Installation de Node.js

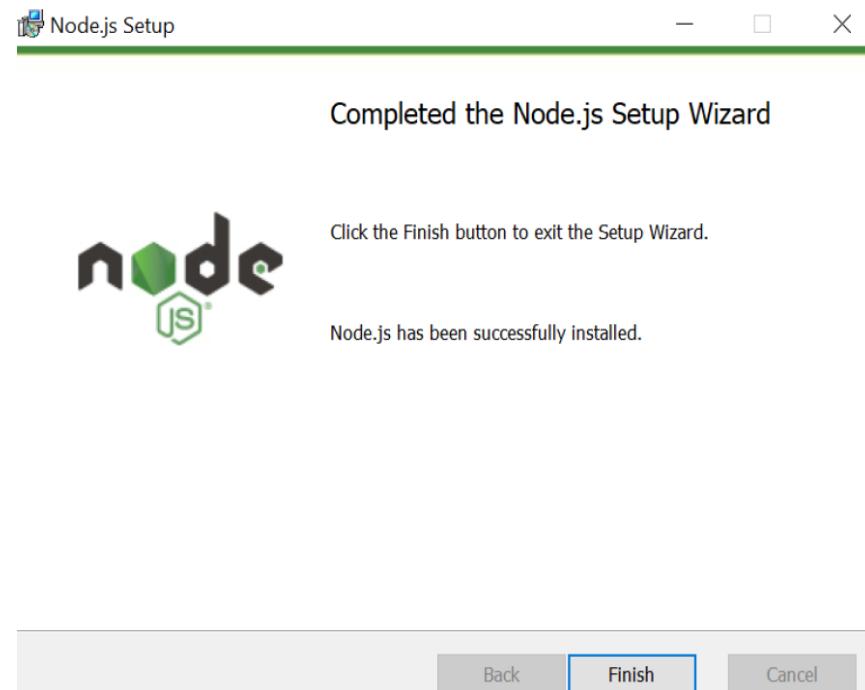
Commencer le processus d'installation: Une fois que vous avez ouvert et exécuté le fichier.msi, le processus d'installation commence.

Node.js vous offre des options pour installer des outils pour les modules natifs. Si vous êtes intéressé par ces derniers, cliquez sur la case à cocher pour marquer vos préférences, ou cliquez sur **Suivant** pour continuer avec la valeur par défaut :



### Installation de Node.js

Le système terminera l'installation en quelques secondes ou minutes et vous montrera un message de réussite. Cliquez sur le bouton Terminer pour fermer le programme d'installation de Node.js.



### Installation de Node.js

#### ■ Vérifier l'installation de Node.js

Pour vérifier l'installation et confirmer que la bonne version a été installée, ouvrez l'invite de commande votre PC et saisissez la commande suivante :

**node -v**

Avec Node, le gestionnaire de paquet NPM sera automatiquement installé

**npm -v**

npm permet de gérer vos dépendances et de lancer vos scripts

```
C:\Users\imane>node -v  
v16.17.1  
  
C:\Users\imane>npm -v  
8.15.0
```

Généralement vous aurez besoin des commandes suivantes :

- ✓ **npm install** : installe le projet sur votre machine.
- ✓ **npm start** : démarre le projet.
- ✓ **npm test** : lance les tests du projet.
- ✓ **npm run dev** : s'occupe de lancer un environnement de développement agréable.



# CHAPITRE 1

## Introduire Express et Node js

1. Rappel du concept des APIs REST ;
2. Rappel des méthodes du protocole http ;
3. Définition de l'ecosystème Node JS ;
4. Configuration de l'environnement de développement;
5. **L'essentiel du Node js**

# Introduire Express et Node js

## *L'essentiel du Node js*



## CHAPITRE 2

### Créer des APIs REST



Ce que vous allez apprendre dans ce chapitre :

 ... heures

## CHAPITRE 3

### Authentifier une API REST avec JWT



Ce que vous allez apprendre dans ce chapitre :

 5 heures



## PARTIE 3

### CRÉER DES MICROSERVICES

- S'initier aux architectures microservices
- Créer une application microservices





# CHAPITRE 1

## S'initier aux architectures microservices

Ce que vous allez apprendre dans ce chapitre :

- Différence entre l'architecture monolithique et l'architecture des microservices ;
- Concepts de base;
- Architecture ;
- Exemples ;

 5 heures



# CHAPITRE 1

## S'initier aux architectures microservices

1. Différence entre l'architecture monolithique et l'architecture des microservices ;
2. Concepts de base;
3. Architecture ;
4. Exemples ;

## 1. Architecture des microservices

### Différence entre l'architecture monolithique et l'architecture des microservices ;

Une application **monolithique** est une application qui est développée en un **seul bloc** (war, jar, Ear, dl ...), avec une même technologie et déployée dans un serveur d'application.

Les difficultés qu'on peut rencontrer avec l'architecture monolithique :

- **Complication du déploiement** : tout changement dans n'importe quelle module de l'application nécessite le redéploiement de toute l'application et menace, par conséquent, son fonctionnement intégral.
- **Scalabilité non optimisée** : La seule façon d'accroître les performances d'une application conçue en architecture monolithique, suite à une augmentation de trafic par exemple, est de la redéployer plusieurs fois sur plusieurs serveurs. Or, dans la majorité des cas, on a besoin d'augmenter les performances d'une seule fonctionnalité de l'application. Mais en la redéployant sur plusieurs serveurs, on accroîtra les performances de toutes les fonctionnalités, ce qui peut être non-nécessaire et gaspiller les ressources de calcul.

➔ Ces deux difficultés principales ont donné naissance à l'architecture **microservices** .

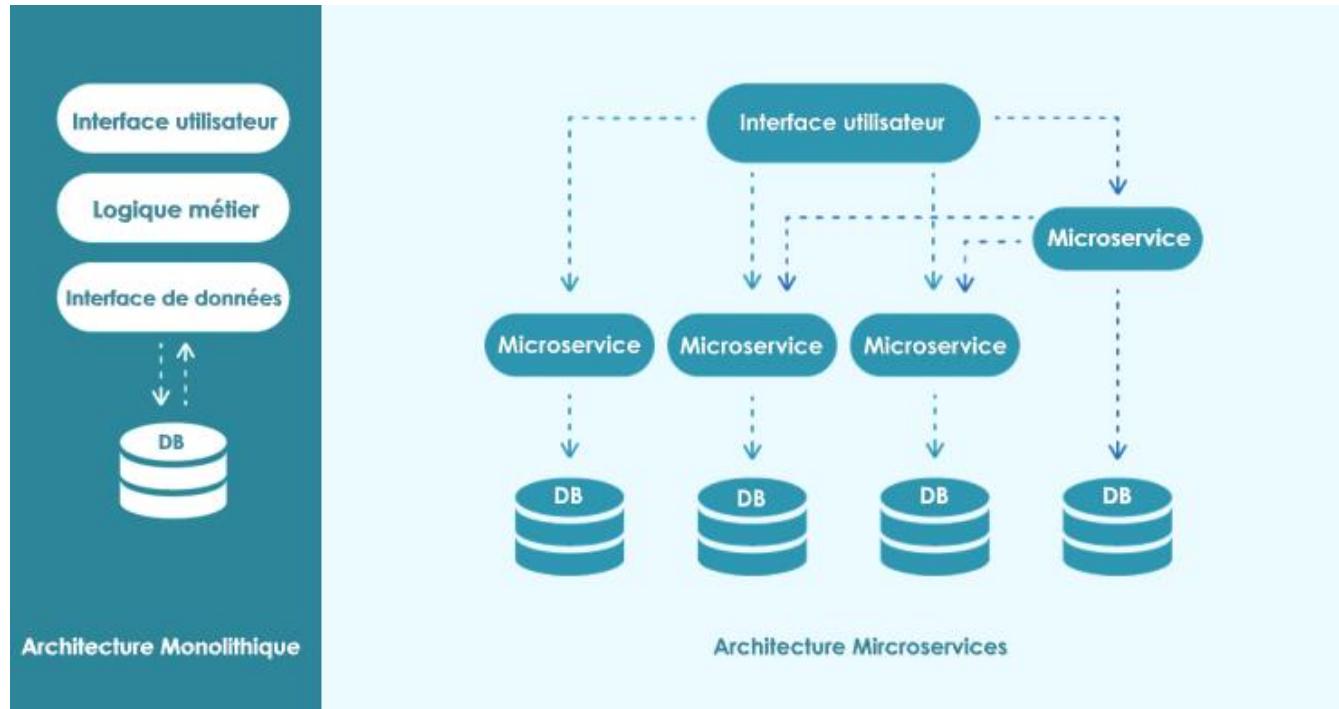
# 1. Architecture des microservices

## Différence entre l'architecture monolithique et l'architecture des microservices ;

### Architecture microservice:

Alors qu'une application monolithique est **une seule unité unifiée**, une architecture microservices la décompose en un ensemble de **petites unités indépendantes**.

Ces unités exécutent chaque processus d'application comme un service distinct. Ainsi, tous les services possèdent leur propre logique et leur propre base de données et exécutent les fonctions spécifiques.



# 1. Architecture des microservices

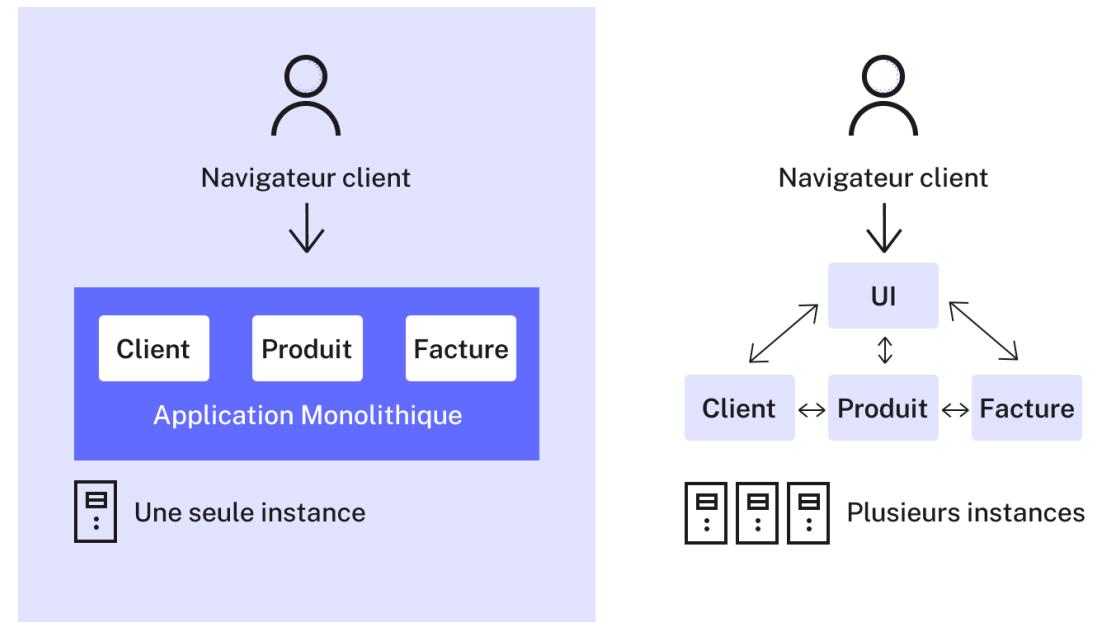
## Différence entre l'architecture monolithique et l'architecture des microservices ;

### Exemple :

Prenons l'exemple d'un site de vente en ligne des produits: l'application est composée de plusieurs modules, à savoir : client, produit et facture.

Pour une application monolithique, ces modules seront déployés dans un seul serveur. En revanche, pour une architecture microservices, chacun de ces modules constituera un service à part avec sa propre base de données. Ces services peuvent être déployés dans des infrastructures différents (sur site, en cloud ...)

Ainsi, un dysfonctionnement du service « facture », par exemple, n'arrêtera pas le fonctionnement de l'ensemble du système.





# CHAPITRE 1

## S'initier aux architectures microservices

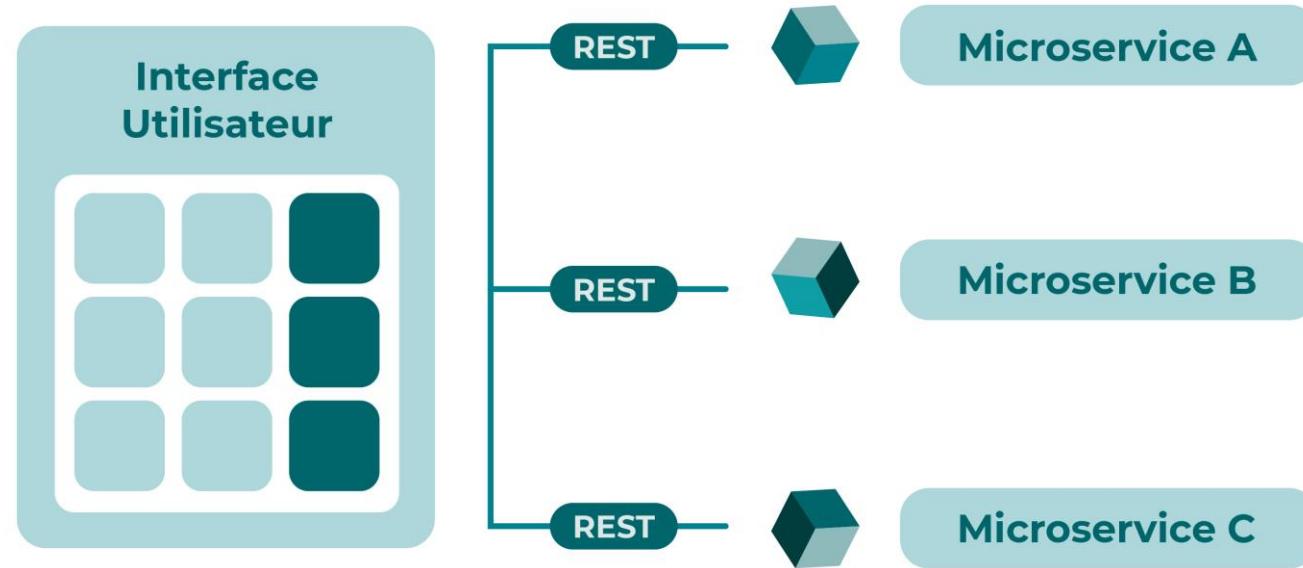
1. Différence entre l'architecture monolithique et l'architecture des microservices ;
- 2. Concepts de base;**
3. Architecture ;
4. Exemples ;

# 1. Architecture des microservices

## Concepts de base

### Principe :

L'architecture Microservices propose une solution en principe simple : **découper** une application en **petits services**, appelés Microservices, **parfaitemnt autonomes**, qui exposent une API que les autres microservices pourront consommer.



Cette application affiche par exemple un produit à vendre. Cette fiche produit est donc constituée par exemple d'une photo, d'un descriptif et d'un prix. Dans le schéma, l'interface utilisateur fait appel à un microservice pour chaque composant à renseigner. Ainsi, celle-ci peut faire une requête REST au **microservice A**, qui s'occupe de la gestion des photos des produits, afin d'obtenir celles correspondant au produit à afficher. De même, les **microservices B et C** s'occupent respectivement des descriptifs et des prix.

# 1. Architecture des microservices

## Concepts de base

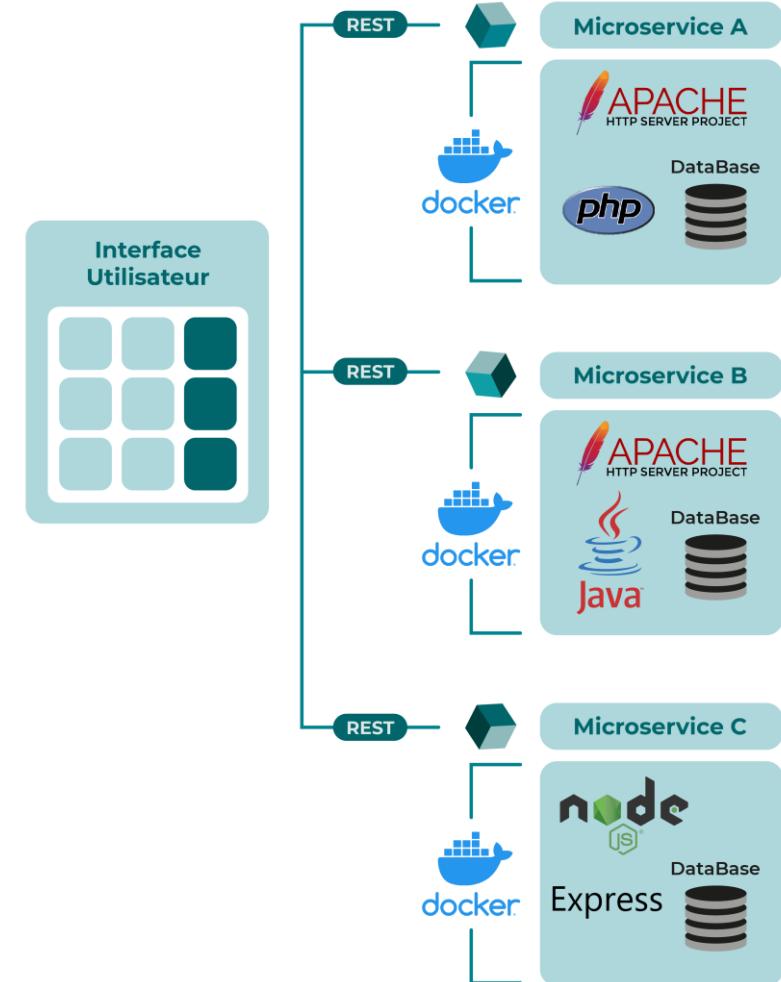
### Agilité technologique :

- Le choix technologique dans une architecture Microservices est totalement ouvert. Il dépend majoritairement des besoins, de la taille des équipes et de leurs compétences et peut être adapté aux besoins spécifiques de chaque micro-service
- Chaque microservice est parfaitement **autonome** : il a sa propre base de données, son propre serveur d'application (Apache, Tomcat, Jetty, etc.), ses propres bibliothèques ...

La plupart du temps, ces microservices sont chacun dans un container [Docker](#), ils sont donc totalement indépendants y compris vis-à-vis de la machine sur laquelle ils tournent.

L'utilisation de conteneurs, permettra ainsi un déploiement continu et rapide des microservices.

Voici un exemple simplifié d'une architecture microservice :



# 1. Architecture des microservices

## Concepts de base



### Le 'Time to market'

- Les microservices peuvent être mis à jour, étendus et déployés indépendamment les uns des autres et par conséquent, beaucoup plus rapidement
- L'indépendance fonctionnelle et technique des microservices permet l'autonomie de l'équipe en charge sur l'ensemble des phases du cycle de vie (développement, tests, déploiement, et exploitation), et favorise par conséquent l'agilité et la réactivité

# 1. Architecture des microservices

## Concepts de base

### Déploiement ciblé :

- Evolution d'une certaine partie sans tout redéployer
- Un seul livrable à partir d'un seul code source
- Moins de coordination entre équipe quand il y a un seul déploiement
  - Plus souvent
  - Moins de risque
  - Plus rapide



**Microservice A**

V2.0



**Microservice B**

V10.2



**Microservice C**

V5.3

# 1. Architecture des microservices

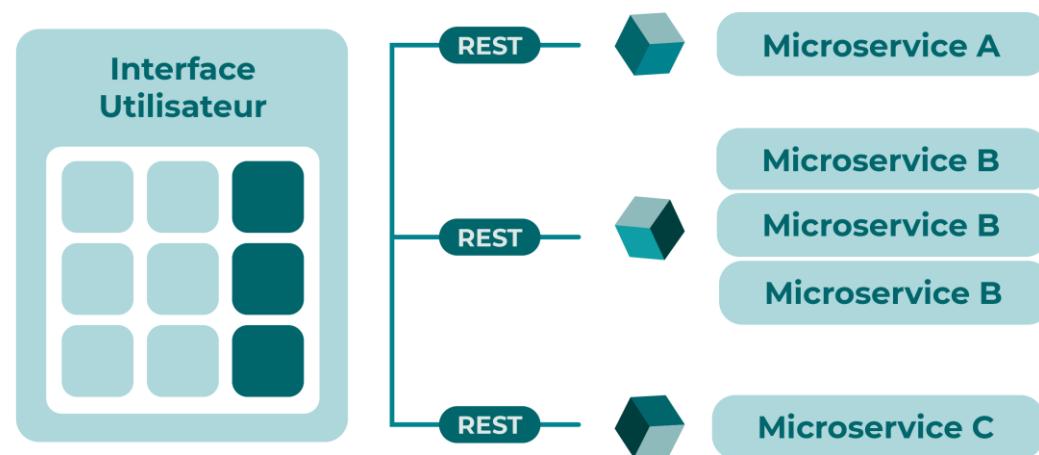
## Concepts de base

### Mise à l'échelle (Scalabilité up/down) :

Selon le trafic du système, on peut ajouter des instances d'un microservice ou en détruire.

Prenons l'exemple de l'événement «Black Friday » de Amazon :

- Les services **produits** et **commandes**, par exemple, seront sollicités plus que les services de **notifications** et **évaluation**.
- Comme chaque microservice est isolée, on a la possibilité de multiplier le nombre d'instances des services les plus demandées , ici : **produits** et **commandes** → Scalabilité up
- A la fin de cet événement, étant donné que le nombre d'utilisateurs diminuera, les instances créées, suite à l'événement de black Friday » peuvent être détruite → Scalabilité down



# 1. Architecture des microservices

## Concepts de base

### Inconvénient de l'architecture microservices :

Le principal inconvénient des microservices est la complexité de tout système distribué.

- **La communication entre les services est complexe** : puisque tout est désormais un service indépendant, vous devez gérer avec soin les demandes transitant entre vos modules. Dans un tel scénario, les développeurs peuvent être obligés d'écrire du code supplémentaire pour éviter toute interruption. Au fil du temps, des complications surviendront lorsque les appels distants subissent une latence.
- **Plus de services équivaut à plus de ressources** : plusieurs bases de données et la gestion des transactions peuvent être pénibles.
- **Les tests globaux sont difficiles** : tester une application basée sur des microservices peut être fastidieux. Dans une approche monolithique, il suffirait de lancer notre WAR sur un serveur d'application et d'assurer sa connectivité avec la base de données sous-jacente.
- **Les problèmes de débogage peuvent être plus difficiles** : chaque service a son propre ensemble de journaux à parcourir. Journal, journaux et autres journaux.
- **Ne s'applique pas à tous les systèmes**
  - Si déploiement fréquent ➔ ok
  - Si déploiement une fois l'an ➔ monolithique

## 1. Architecture des microservices

### Concepts de base



Qui utilisent les microservices ?

Uber



NETFLIX

eBay

amazon



# CHAPITRE 1

## S'initier aux architectures microservices

1. Différence entre l'architecture monolithique et l'architecture des microservices ;
2. Concepts de base;
- 3. Architecture ;**
4. Exemples ;

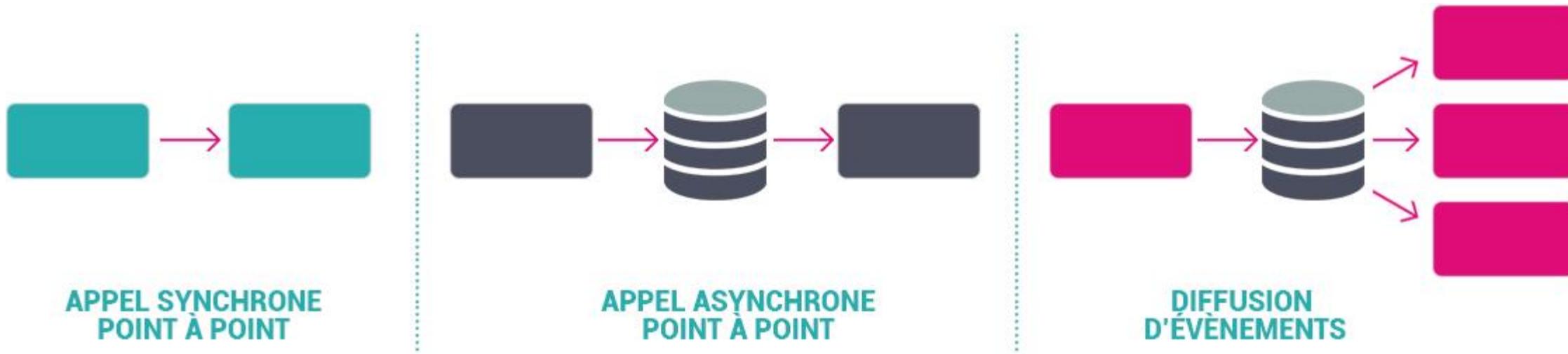
# 1. Architecture des microservices

## Architecture

### Communication entre services:

On distingue trois types de modes de communication entre services:

- Appel synchrone point à point
- Diffusion de messages asynchrones point à point
- Diffusion d'événements



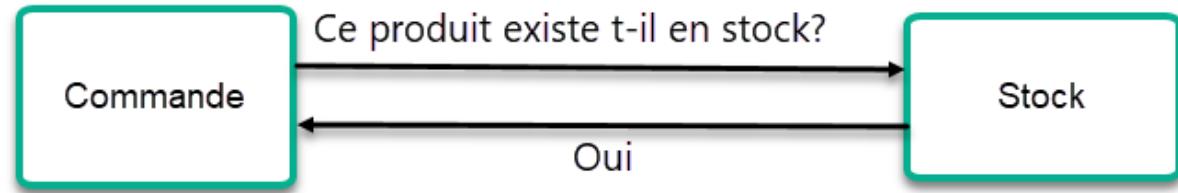
Ces patterns sont tous **complémentaires** entre eux et ont, d'une manière ou d'une autre, leur place dans les architectures Microservices que nous concevons. Cependant, le choix au cas par cas d'un mode de communication ou d'un autre, est loin d'être anodin et devrait être réalisé en connaissance de cause

# 1. Architecture des microservices

## Architecture

### Types de communication entre services: Appel synchrone point à point

- Un service appelle un autre service et attend une réponse
- ➔ Ce type de communication est utilisé lorsqu'un service émetteur a besoin de la réponse pour continuer son processus

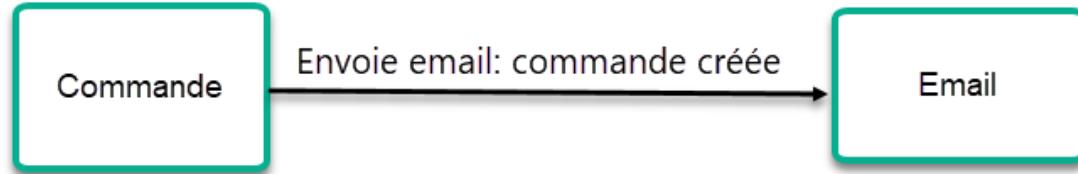


# 1. Architecture des microservices

## Architecture

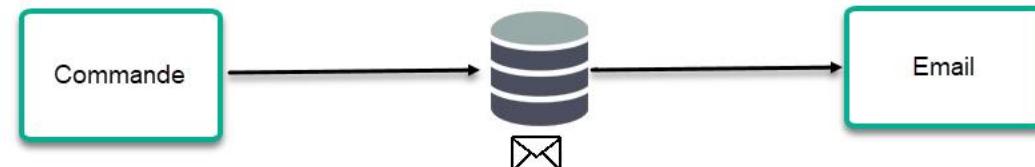
### Types de communication entre services: Appel asynchrone point à point

- Un service appelle un autre service et continue son processus
  - Le service émetteur n'attend pas de réponse: Fire and forget
- ➔ Ce type de communication est utilisé lorsqu'un service désire envoyer un message à un autre service



### Implémentation:

Pour exploiter ce type de communication, on emploie un protocole de transport de messages (AMQP)

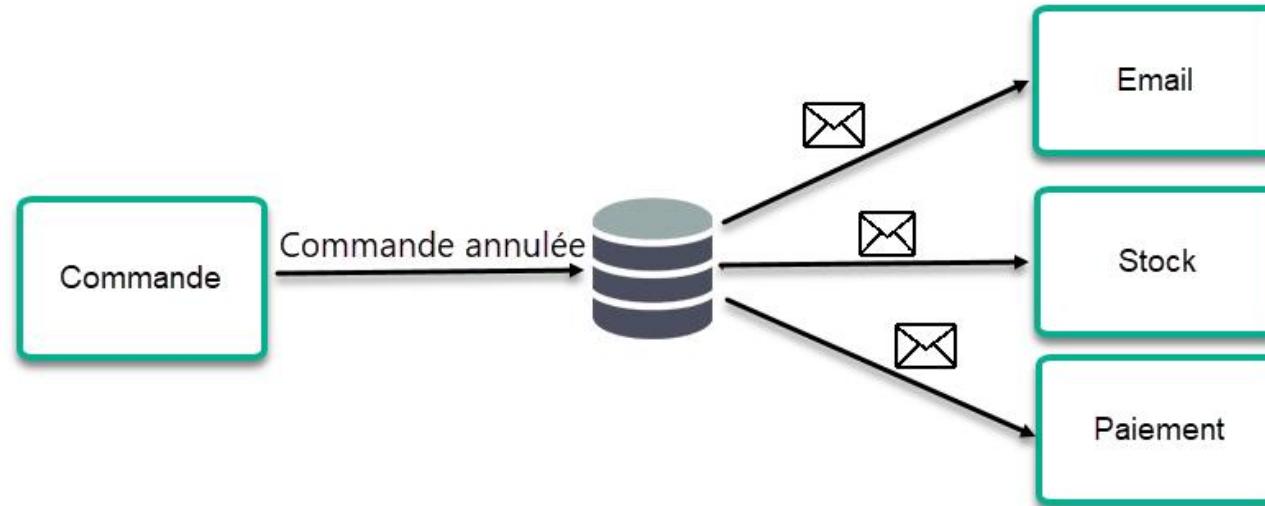


# 1. Architecture des microservices

## Architecture

### Types de communication entre services: Diffusion d'événements

- Quand un service désire envoyer une notification aux autres services
  - il n'a aucune idée des services écoutant cet événement (listen)
  - Il n'attend pas de réponse: Fire and forget
- ➔ Ce type de communication est utilisé quand un service veut notifier tout le système par un évènement

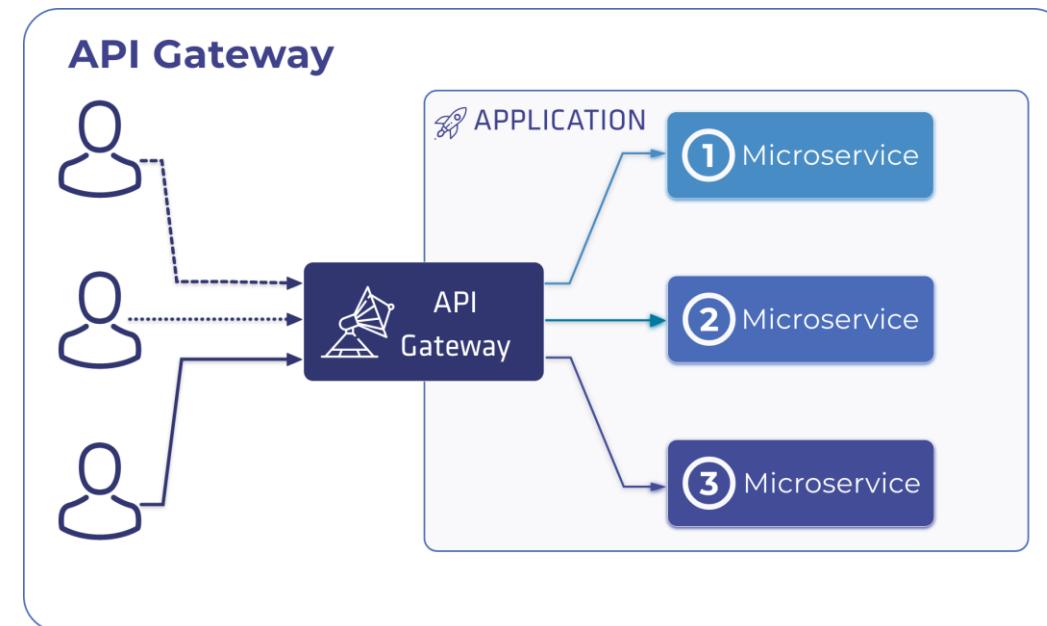


# 1. Architecture des microservices

## Architecture

### Intermédiaire entre client et microservices:

- Les passerelles API, ou API gateways:
  - constituent **la couche intermédiaire** entre les microservices et les applications clientes qui ont besoin des microservices pour fonctionner.
  - servent **de point d'entrée unique** à notre système et n'exposent que **les points de terminaison requis**.
  - **équilibrivent** également la **charge** des demandes des clients et les répartissent sur plusieurs instances d'un service.





# CHAPITRE 1

## S'initier aux architectures microservices

1. Différence entre l'architecture monolithique et l'architecture des microservices ;
2. Concepts de base;
3. Architecture ;
- 4. Exemples ;**

## 1. Architecture des microservices

### Etude de cas

On souhaite mettre en place une architecture microservice pour un système de **gestion des emprunts d'une librairie**.

L'objectif de cet exemple est d'identifier les services composant ce système ;

Chaque service doit être :

- *défini clairement*
- *autonome*
- *possède un canal de communication claire*

#### Gestion des emprunts d'une librairie:

Le système offrira les fonctionnalités suivantes:

- Gestion des livres
- Gestion des emprunts
- Gestion des clients
- Envoi des notification (Exemple: Quand un client indique qu'il a besoin d'un livre, le système lui enverra une notification une fois ce livre est retourné)
- Paiement

# 1. Architecture des microservices

## Etude de cas



### Gestion d'une librairie:

On peut distinguer 3 services métier et 3 utilités :

#### Service Métier

Service  
Livre

Service  
Emprunt

Service  
Client

#### Utilités

Notification

Paiement (Service  
Externe)

View

# 1. Architecture des microservices

## *Etude de cas*

### Gestion d'une librairie:

Le service « **Livre** » :

- Permet de gérer le stock des livres de la librairie
- Est utilisé par le bibliothécaire
- Dispose d'une base de données
- Doit être synchrone (réponse immédiate) : Si le bibliothécaire, par exemple, ajoute un nouveau livre, ce dernier doit immédiatement savoir si le livre a été ajouté avec succès ou non.
- Ne dépend pas des autres services (On n'a pas besoin des autres services pour que ce service fonctionne correctement)
- Expose une API REST :

Service  
Livre

Fonctionnalité	Méthode / Chemin	Code retourné
Retourner les informations d'un livre donné	GET /api/v1/livre/{idLivre}	200 OK 404 Non trouvé
Ajouter un nouveau livre	POST /api/v1/livre	200 OK
Modifier un livre	PUT /api/v1/livre/{idLivre}	200 OK 404 Non trouvé
Supprimer un livre	DELETE /api/v1/livre/{idLivre}	200 OK 404 Non trouvé

# 1. Architecture des microservices

## Etude de cas

### Gestion d'une librairie:

Le service « Emprunt » :

- Permet de gérer les emprunts des livres
- Est utilisé par le bibliothécaire
- Dispose d'une base de données
- Doit être synchrone (réponse immédiate)
- Ne dépend pas des autres services (Même s'il fait référence aux bases de données des services client et livre mais il ne dépend pas d'eux : Si le service livre, par exemple, ne fonctionne pas, le service emprunt va continuer son fonctionnement car il utilise l'identifiant du livre et non pas le service livre)
- Expose une API REST :

**Service  
Emprunt**

Fonctionnalité	Méthode / Chemin	Code retourné
Ajouter un nouveau emprunt	POST /api/v1/emprunt	200 OK
Retourner un livre	POST /api/v1/emprunt	200 OK
Retourner les emprunts d'un client donné	GET /api/v1/emprunt/{idClient}	200 OK 404 Non trouvé

# 1. Architecture des microservices

## Etude de cas



### Gestion d'une librairie:

Le service « **Notification** » :

- Est utilisé pour envoyer des notifications aux clients de la librairie  
(Ex. un livre a été retourné, nouveaux livres ont été ajoutés ...)
- Il est utilisé par les autres services (et non pas par des humains)  
(Ex. Le service Livre peut l'utiliser quand un livre est ajouté, ou le service Emprunt peut l'employer au cas d'un retour)
- Asynchrone : Les services appelant n'attendent pas une réponse immédiate)  
(Ex. Le service Livre veut envoyé une notification à 1000 clients suite à un ajout d'un nouveau livre)
- La communication est assuré par le biais des « Queues »: il s'agit **d'une diffusion d'événement**

Service  
Notification

# 1. Architecture des microservices

## *Etude de cas*



### Gestion d'une librairie:

Le service « **Paiement** » :

- Est utilisé pour envoyer des instructions de paiement à des services externes
- Est utilisé par les autres services

(Ex. Quand un nouveau client s'inscrit à la librairie, il doit payer sa facture annuelle, ainsi le service client appelle le service paiement)

- Asynchrone (Réponse immédiate n'est pas obligatoire)
- La communication est assuré par le biais des « Queues »:

(Si le service client, par exemple, tombe en panne, le message de paiement va résider dans la queue des messages jusqu'à sa consommation par le service paiement)

Service  
Paiement

# 1. Architecture des microservices

## *Etude de cas*



### Gestion d'une librairie:

Le service « **View** » :

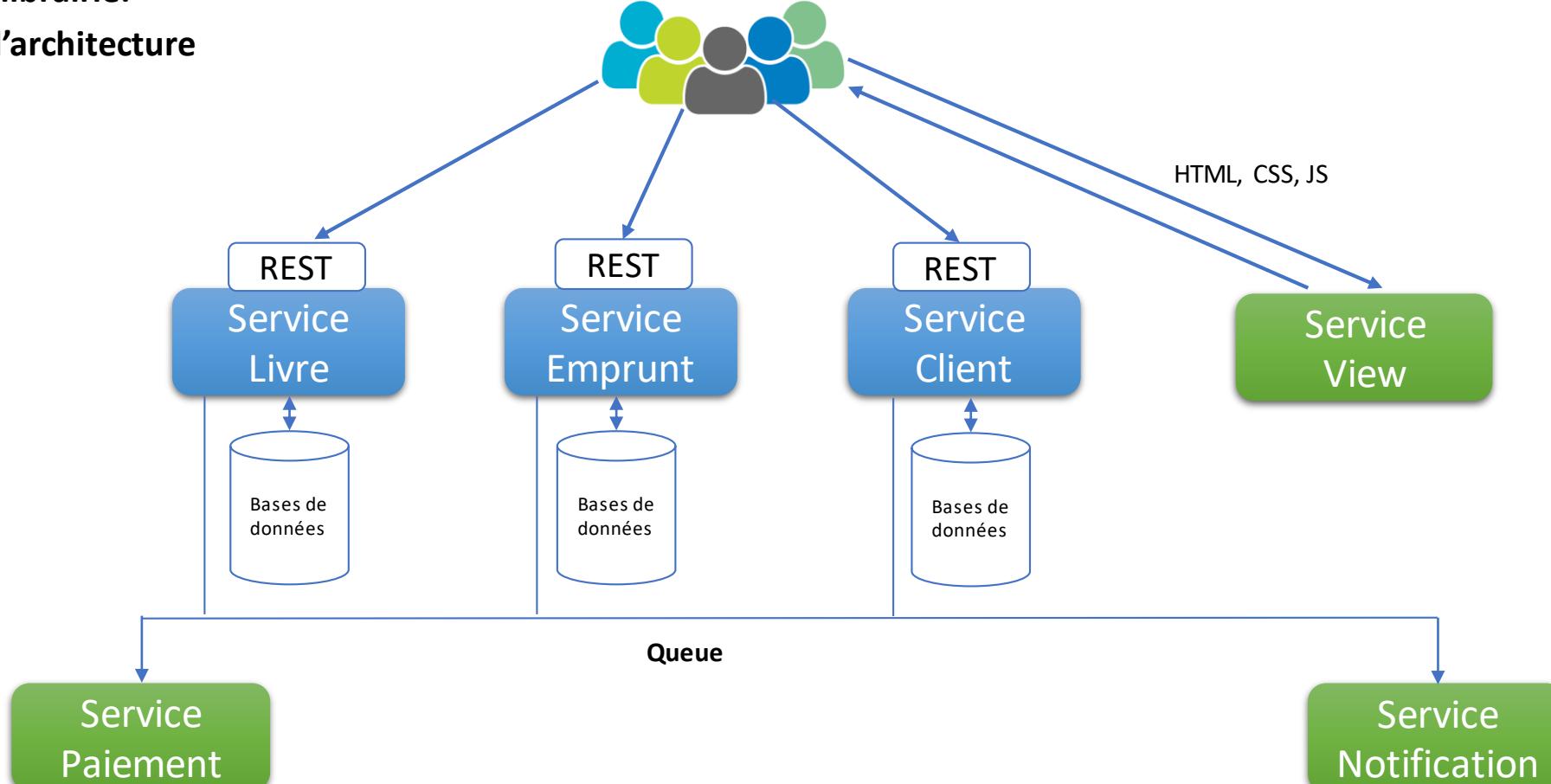
- Sert du contenu statique aux navigateurs : HTML, CSS, JS
- Est un service basique sans logique ni code
- Ne nécessite pas d'API,
- Il interfère directement avec les navigateurs

Service  
View

# 1. Architecture des microservices

## Etude de cas

### Gestion d'une librairie: Diagramme de l'architecture

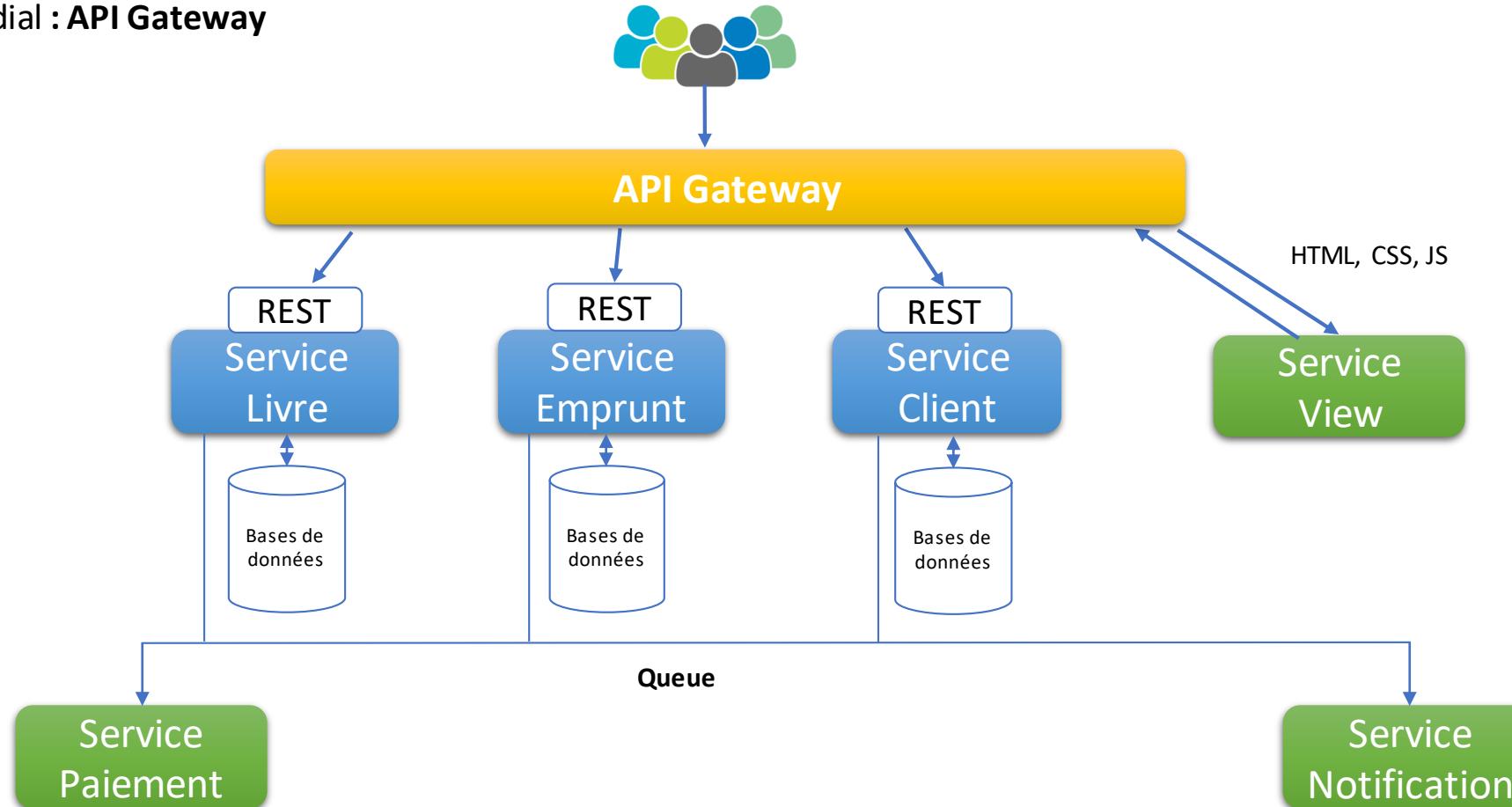


# 1. Architecture des microservices

## Etude de cas

### Gestion d'une librairie:

Afin d'orchestrer les requêtes aux microservices et de les séparer les utilisateurs finaux, une implémentation d'un service intermédiaire s'avère primordial : **API Gateway**





## CHAPITRE 2

### Créer des microservices

Ce que vous allez apprendre dans ce chapitre :

- Microservice versus API REST ;
- Création d'une application en microservices avec Node js ;
- Communication entre microservices avec Rabbitmq :
  - Principe de base de RabbitMQ;
  - Installation du serveur Rabbitmq
  - Installation du module client amqplib avec npm ;
  - Mise en œuvre de la communication
- Mise en place d'un reverse proxy en utilisant Nginx :
  - Rôle d'un reverse proxy
  - Installation et configuration de Nginx



heures



## CHAPITRE 2

### Créer des microservices

- 1. Microservice versus API REST**
2. Création d'une application en microservices avec Node js
3. Communication entre microservices avec Rabbitmq
4. Mise en place d'un reverse proxy en utilisant Nginx

## 2. Créer des microservices

### *Microservice versus API REST*



#### Qu'est ce qu'un API?

- API signifie Application Programming Interface;
- C'est une **interface** qui permet à deux applications de communiquer;
- Un logiciel est utilisé par des **utilisateurs** via une **interface utilisateur**. Cependant, les gens ne sont pas les seuls à utiliser les logiciels. Le logiciel est également utilisé par d'autres applications. Cela nécessite un autre type d'interface qui s'appelle une interface de programmation d'application, ou en abrégé **API**;
- Ils sont considérés comme des **portes** qui permettent aux **développeurs** d'interagir avec une application;
- C'est une façon d'effectuer des requêtes sur un composant.

#### Exemples:

- API Google Map permet aux développeurs de générer et d'afficher des cartes personnalisables sur leur site web.

## 2. Créer des microservices

### Microservice versus API REST



#### Tableau de comparaison

	API	Microservice
Définition	Il s'agit d'un ensemble de méthodes prédéfinies facilitant la <b>communication</b> entre différents composants.	C'est une architecture qui consiste à <b>diviser</b> les différentes fonctions d'une application en composants <b>plus petits et plus agiles</b> appelés services.
Concept	Les API offrent un moyen simple de se connecter, de <b>s'intégrer</b> et <b>d'étendre</b> un système logiciel.	L'architecture des microservices est généralement organisée autour des <b>besoins en capacité et des priorités</b> de l'entreprise;
Fonction	Les API fournissent <b>une interface réutilisable</b> à laquelle différentes applications peuvent se connecter facilement.	Les microservices <b>s'appuient largement sur les APIs et les passerelles API</b> pour rendre possible la communication entre les différents services.



## CHAPITRE 2

### Créer des microservices

1. Microservice versus API REST
2. **Création d'une application en microservices avec Node js**
3. Communication entre microservices avec Rabbitmq
4. Mise en place d'un reverse proxy en utilisant Nginx

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js*



#### Raisons de créer des microservices avec NodeJS

Parmi tous les langages de programmation utilisés dans le développement d'applications de microservices, NodeJS est largement utilisé par une grande communauté des développeurs pour ses caractéristiques et les avantages qu'il apporte.

Voici quelques raisons pour lesquelles les microservices avec Node.JS sont le meilleur choix :

- **Il améliore le temps d'exécution** puisque NodeJS s'exécute sur le moteur V8 de Google, compile la fonction en code machine natif et effectue également des opérations CPU et IO à faible latence.
- **L'architecture événementielle** de NodeJS le rend très utile pour développer des applications événementielles.
- Les bibliothèques NodeJS prennent en charge **les appels non bloquants** qui continuent de fonctionner sans attendre le retour de l'appel précédent.
- Les applications créées avec NodeJS sont **évolutives**, ce qui signifie que le modèle d'exécution prend en charge la mise à l'échelle en attribuant la demande à d'autres threads de travail.

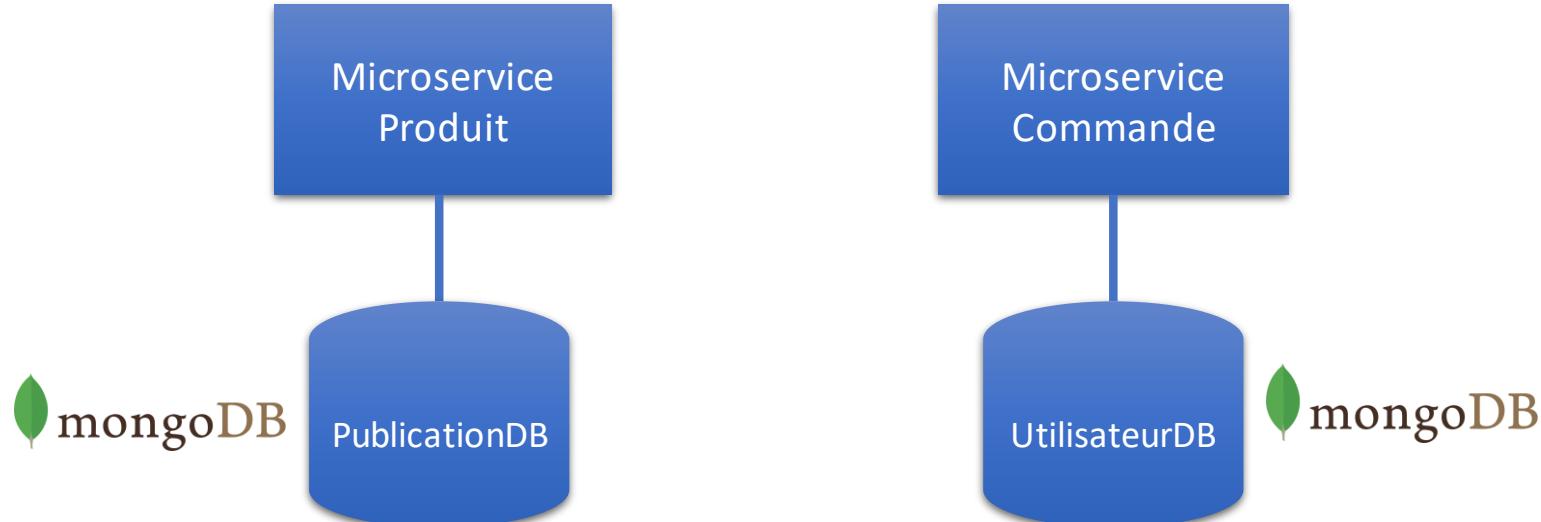
## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple*

Un microservice est en outre défini par son architecture et sa portée. Par conséquent, d'un point de vue technique, implémenter un microservice via HTTP ne serait pas différent de l'implémentation d'une API NodeJS.

#### Exemple de microservices créés en Node js (Communication synchrone):

- Soit une simple partie d'application e-commerce gérant les produits et leurs commandes;
- L'application sera décomposée, dans un premier temps, en deux microservices : produit et commande;
- Chaque service aura sa propre base de données sous MongoDB
- Les deux services seront considérés comme deux applications séparées, chacune expose son API et écoute sur son propre port;



## 2. Créer des microservices

*Création d'une application en microservices  
avec Node js : Exemple*



APIs à créer pour cet exemple :

Microservice  
Produit

Le port d'écoute : **4000**

Méthode	URL	Signification
GET	localhost:4000/produit/acheter	Cherche les produits à acheter et retourne leur objets correspondants
POST	localhost:4000/produit/ajouter	Ajoute un nouveau produit à la base de données

Microservice  
Commande

Le port d'écoute : **4001**

Méthode	URL	Signification
POST	localhost:4001/commande/ajouter	Ajoute une nouvelle commande à la base de données

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple*



#### Données de l'exemple :

##### - Structure d'un objet produit :

**\_id** : son identifiant

**nom** : son nom

**description** : des informations le décrivant

**prix** : sa valeur

**created\_at** : la date de sa création (par défaut elle prend la date système)

##### - Structure d'un objet commande :

**\_id** : son identifiant

**produits**: un tableau regroupant les ids des produits concernés par cette commande

**email\_utilisateur**: l'adresse email de celui à qui appartient la commande

**prix\_total**: le total des prix à payer pour finaliser la commande

**created\_at**: la date de sa création (par défaut elle prend la date système)

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple*



#### Exemple:

Afin de réaliser cet exemple, suivons les étapes ci-après:

1. Créer un dossier appelé expMicroservice;
2. Sous le dossier expMicroservice, créer deux sous dossiers : produit-service et commande-service;

#### 3. Produit-service :

- a. Sous le dossier produit-service, ouvrir le terminal et exécuter les commandes suivantes :

```
npm init -y
```

```
npm install express mongoose nodemon
```

- b. Ajouter à la racine de ce dossier un fichier vide "index.js";

- c. Modifier la partie script du fichier package.json et remplacer le script existant par :

```
"start": "nodemon index.js"
```

The screenshot shows a code editor with two tabs: 'EXPLORER' and 'package.json'. The EXPLORER tab shows a project structure with a root folder 'EXPMICROSERVICE' containing 'publication-service' and 'utilisateur-service'. Inside 'publication-service', there are 'node\_modules', 'index.js', 'package-lock.json', and 'package.json'. The 'package.json' file is selected and shown in the code editor.

```
1 {  
2   "name": "publication-service",  
3   "version": "1.0.0",  
4   "description": "",  
5   "main": "index.js",  
6   "scripts": {  
7     "start": "nodemon index.js"  
8   },  
9   "keywords": [],  
10  "author": "",  
11  "license": "ISC",  
12  "dependencies": {  
13    "express": "^4.18.2",  
14    "mongoose": "^6.8.0",  
15    "nodemon": "^2.0.20"  
16  }  
17 }
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple*



#### Exemple:

#### 3. Produit-service :

d. Ajouter un fichier à la racine appelé « Produit.js »;

Ce fichier représentera le modèle « produit » à créer à base d'un schéma de données **mongoose**;

Ce schéma permettra par la suite de créer une collection appelée « produit » sous une base de données **MongoDB**;

```
const mongoose = require("mongoose");

const ProduitSchema = mongoose.Schema({
  nom: String,
  description: String,
  prix: Number,
  created_at: {
    type: Date,
    default: Date.now(),
  },
});

module.exports = Produit = mongoose.model("produit", ProduitSchema);
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple*



#### e. Modifier le fichier index.js :

```
const express = require("express");
const app = express();
const PORT = process.env.PORT_ONE || 4000;
const mongoose = require("mongoose");
const Produit = require("./Produit");

app.use(express.json());

//Connection à la base de données MongoDB « publication-service-db »
//(Mongoose créera la base de données s'il ne le trouve pas)
mongoose.set('strictQuery', true);
mongoose.connect(
  "mongodb://localhost/produit-service",
  {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  },
  () => {
    console.log(`Produit-Service DB Connected`);
  }
);
app.post("/produit/ajouter", (req, res, next) => {
  const { nom, description, prix } = req.body;
  const newProduit = new Produit({
    nom,
    description,
    prix
  });
  //La méthode save() renvoie une Promise.
  //Ainsi, dans le bloc then(), nous renverrons une réponse de réussite.
  //Dans le bloc catch(), nous renverrons une réponse avec l'erreur générée par Mongoose ainsi qu'un code d'erreur 400.
  newProduit.save()
    .then(produit => res.status(201).json(produit))
    .catch(error => res.status(400).json({ error }));
});

app.get("/produit/acheter", (req, res, next) => {
  const { ids } = req.body;
  Produit.find({ _id: { $in: ids } })
    .then(produits => res.status(201).json(produits))
    .catch(error => res.status(400).json({ error }));
});

app.listen(PORT, () => {
  console.log(`Product-Service at ${PORT}`);
});
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple*



#### 4. Tester l'API exposée par le service : Publication-service :

- On peut démarrer le service publication en tapant: npm start (et l'arrêter en tapant ctr+c)
- A l'aide de Postman, Créer une requête POST en utilisant l'URL **localhost:4000/publication/create**
- Sous « Body », choisir l'option « row » et JSON comme type de données
- Dans la zone de texte en bas, taper un objet JSON avec un titre et un contenu :
- Après l'envoi de cette requête, on peut obtenir le résultat ci-après :

The screenshot shows a Postman interface with the following details:

- Method: POST
- URL: localhost:4000/produit/ajouter
- Body tab selected, type: JSON
- Body content:

```
1 {  
2   "nom": "Fairy",  
3   "description": "Produit de nettoyage",  
4   "prix": 130  
5 }
```
- Response status: 201 Created
- Response time: 93 ms
- Response size: 385 B
- Response content (Pretty):

```
1 {  
2   "nom": "Fairy",  
3   "description": "Produit de nettoyage",  
4   "prix": 130,  
5   "created_at": "2022-12-17T07:58:50.785Z",  
6   "_id": "639d76c1135d56ac3f30295d",  
7   "__v": 0  
8 }
```

## 2. Créer des microservices

### Création d'une application en microservices avec Node js : Exemple

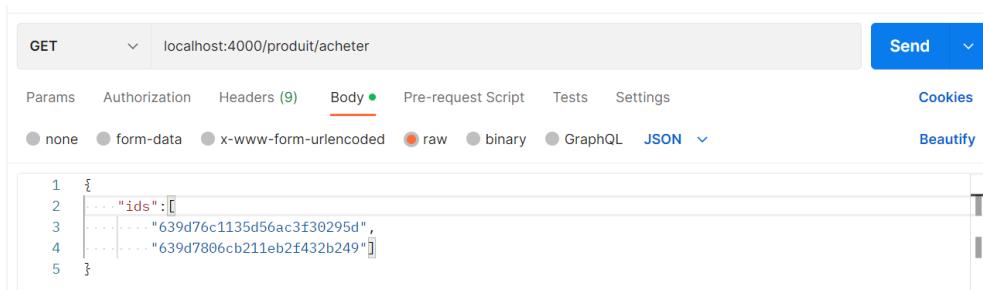
- On suppose qu'on enregistré dans la collection Produit les deux produits suivants:



```
_id: ObjectId('639d76c1135d56ac3f30295d')
nom: "Fairy"
description: "Produit de nettoyage"
prix: 130
created_at: 2022-12-17T07:58:50.785+00:00
__v: 0

_id: ObjectId('639d7806cb211eb2f432b249')
nom: "Clear"
description: "Après shampoing"
prix: 95
created_at: 2022-12-17T08:02:29.147+00:00
__v: 0
```

- Si on possède leur ids et on souhaite avoir plus de détails afin de les acheter, on doit appeler la méthode de l'API **localhost:4000/produit/acheter** en lui passant les deux ids comme paramètre :

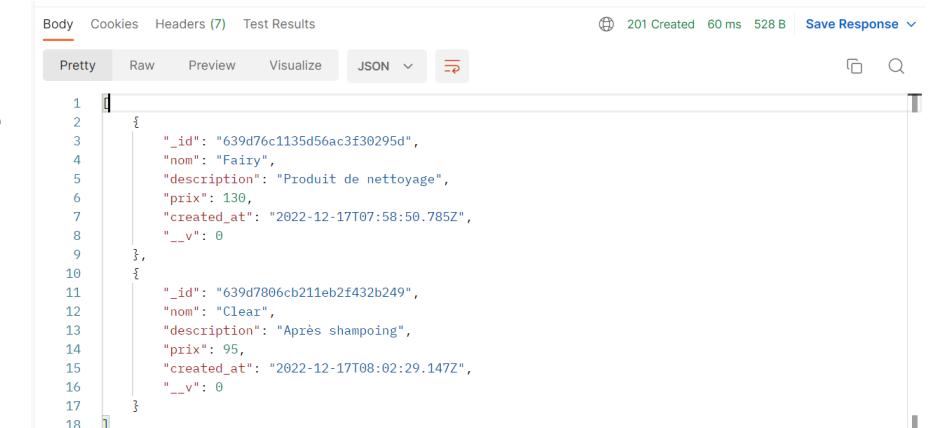


GET localhost:4000/produit/acheter

Body (JSON)

```
1 {
2   ...
3   "ids": [
4     "639d76c1135d56ac3f30295d",
5     "639d7806cb211eb2f432b249"
6   ]
7 }
```

Voici le résultat de  
la requête :



Body (Pretty)

```
1 [
2   {
3     "_id": "639d76c1135d56ac3f30295d",
4     "nom": "Fairy",
5     "description": "Produit de nettoyage",
6     "prix": 130,
7     "created_at": "2022-12-17T07:58:50.785Z",
8     "__v": 0
9   },
10  {
11    "_id": "639d7806cb211eb2f432b249",
12    "nom": "Clear",
13    "description": "Après shampoing",
14    "prix": 95,
15    "created_at": "2022-12-17T08:02:29.147Z",
16    "__v": 0
17  }
18 ]
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple*



#### 4. commande-service :

- a. Sous le dossier « commande-service », refaire les mêmes étapes a, b et c de la création du service produit
- b. Ajouter le fichier Commande.js, à la racine de ce dossier, permettant de créer le schéma de la collection « commande » :

```
const mongoose = require("mongoose");

const CommandeSchema = mongoose.Schema({
    produits: {
        type: [String]
    },
    email_utilisateur: String,
    prix_total: Number,
    created_at: {
        type: Date,
        default: Date.now(),
    },
});

module.exports = Commande = mongoose.model("commande", CommandeSchema);
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple*



#### 4. commande-service :

- c. Pour ajouter une commande, il faut fournir les identifiants des produits à commander et de récupérer le prix de chacun;

Pour ce, le service commande enverra une requête http, au service produit, pour avoir plus de détails de chaque produit en se basant sur son id;

**Axios** est un client HTTP basé sur des promesses pour le navigateur et Node.js. **Axios** facilite l'envoi de requêtes HTTP asynchrones aux points de terminaison REST et l'exécution d'opérations CRUD.

Pour utiliser Axios, il faut tout d'abord l'installer via la commande : **npm i axios**

Ajouter, à la racine, un fichier index.js avec le contenu ci-après :

```
const express = require("express");
const app = express();
const PORT = process.env.PORT_ONE ||
4001;
const mongoose = require("mongoose");
const Commande = require("./Commande");
const axios = require('axios');

//Connexion à la base de données
mongoose.set('strictQuery', true);
mongoose.connect(
  "mongodb://localhost/commande-
  service",
  {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  },
  () => {
    console.log(`Commande-Service DB
Connected`);
  }
);
app.use(express.json());
//....
```

## 2. Créer des microservices

### Création d'une application en microservices avec Node js



... Suite de index.js

```
//Calcul du prix total d'une commande en passant en paramètre un tableau des produits
function prixTotal(produits) {
    let total = 0;
    for (let t = 0; t < produits.length; ++t) {
        total += produits[t].prix;
    }
    console.log("prix total :" + total);
    return total;
}

//Cette fonction envoie une requête http au service produit pour récupérer le tableau des produits qu'on désire commander (en se basant sur leurs ids)
async function httpRequest(ids) {
    try {
        const URL = "http://localhost:4000/produit/acheter"
        const response = await axios.post(URL, { ids: ids }, {
            headers: {
                'Content-Type': 'application/json'
            }
        });
        //appel de la fonction prixTotal pour calculer le prix total de la commande en se basant sur le résultat de la requête http
        return prixTotal(response.data);
    } catch (error) {
        console.error(error);
    }
}
app.post("/commande/ajouter", async (req, res, next) => {
    // Crédit d'une nouvelle commande dans la collection commande
    const { ids, email_utilisateur } = req.body;
    httpRequest(req.body.ids).then(total => {
        const newCommande = new Commande({
            ids,
            email_utilisateur: email_utilisateur,
            prix_total: total,
        });
        newCommande.save()
            .then(commande => res.status(201).json(commande))
            .catch(error => res.status(400).json({ error }));
    });
});

app.listen(PORT, () => {
    console.log(`Commande-Service at ${PORT}`);
});
```

## 2. Créer des microservices

# *Création d'une application en microservices avec Node.js : Exemple*



Afin de tester l'appel de la méthode d'API **commande/ajouter**, il faut démarrer les deux services :

```
asmae@DESKTOP-PGQ5OJJ MINGW64
               /expMicroservice/produi
t-service
$ npm start

> produit-service@1.0.0 start
> nodemon index.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`*
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
Product-Service at 4000
Produit-Service DB Connected

asmae@DESKTOP-PGQ5OJJ MINGW64
               /expMicroservice/comman
de-service
$ npm start

> commande-service@1.0.0 start
> nodemon index.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`*
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node index.js'
Commande-Service at 4001
Commande-Service DB Connected
```

Sous Postman, lancer et exécuter la requête POST en lui passant deux paramètres: un tableau des ids, et un email\_utilisateur

The screenshot shows the Postman application interface. At the top, there's a header bar with 'POST' selected as the method, the URL 'localhost:4001/commande/ajouter', and a 'Send' button on the right. Below the header, there are tabs for 'Params', 'Authorization', 'Headers (9)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. Under 'Body', the 'JSON' tab is selected, and the raw JSON payload is displayed:

```
1 "email_utilisateur": "ahmed.alaoui@gmail.com",
2 "ids": [
3     "639d76c1135d56ac3f30295d",
4     "639d7896cb211eb2f432b249"
5 ]
6
```

Below the body editor, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. On the right side, status information is shown: '201 Created', '42 ms', '452 B', and a 'Save Response' button. At the bottom, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and another 'JSON' dropdown.

## 2. Créer des microservices

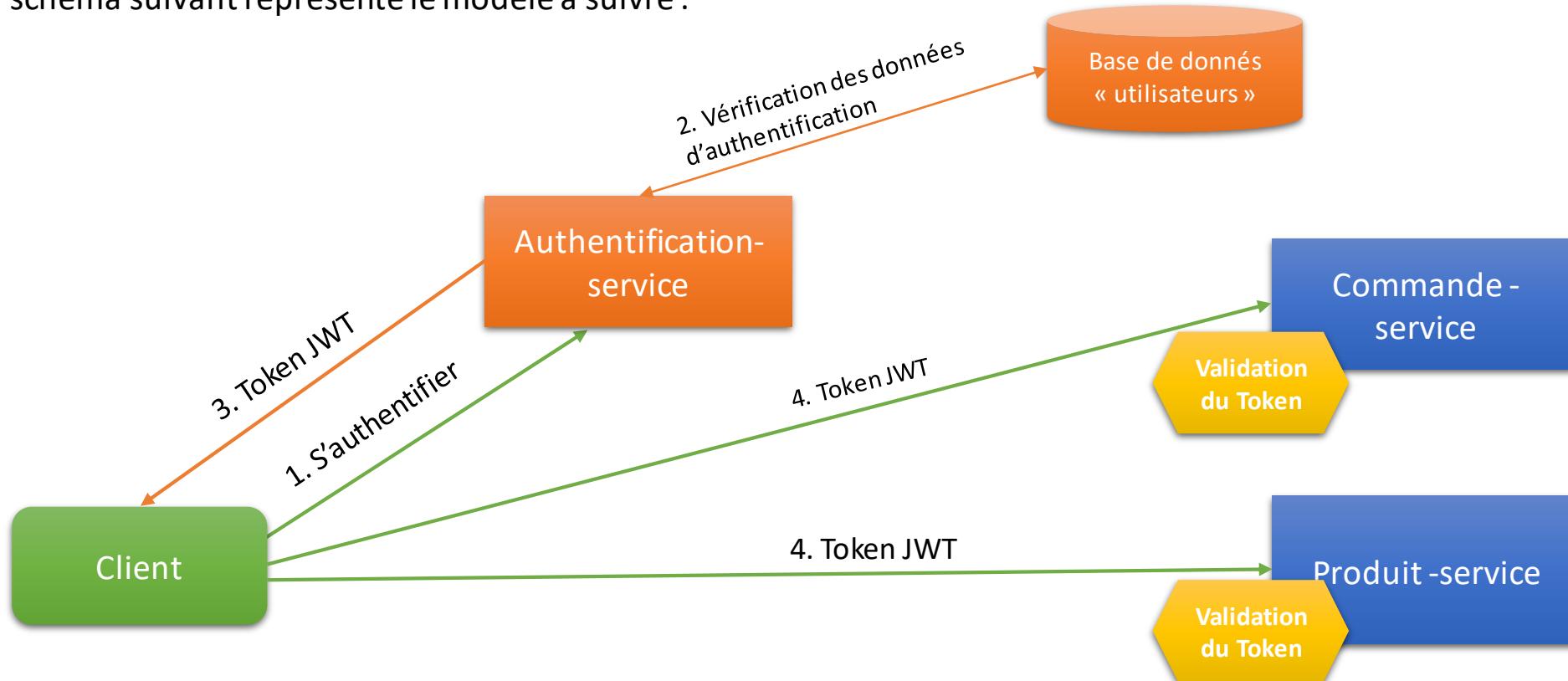
### *Création d'une application en microservices avec Node.js : Exemple (Authentification)*

#### Authentification :

Avant de pouvoir exploiter les différentes méthodes des API créées, un utilisateur doit être connecté.

Ainsi, on doit ajouter une étape intermédiaire d'authentification avant de consulter n'importe quel service ;

Le schéma suivant représente le modèle à suivre :



## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple (Authentification)*



#### Authentification :

1. Le client envoie premièrement une requête au service d'authentification; En cas de réussite, celui-là crée un token JWT et l'envoie au client;

*L'utilisation de JWT dans les microservices peut être comprise comme des passeports, des clés, des signes d'identification, ... pour savoir qu'il s'agit d'une demande avec une origine valide.*

2. Le client sert du token pour demander l'autorisation auprès des services à exploiter.
3. Avant de répondre aux demandes auprès des utilisateurs, les services vérifient la validité du token.

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple (Authentification)*



#### Authentification-service :

Ajoutons un nouveau service appelé « auth-service »:

1. Sous le dossier « exMicroservice », créer un sous dossier « auth-service »;
2. Initialiser npm et exécuter sous le terminal : **npm install express nodemon mongoose jsonwebtoken bcryptjs**
3. Modifier le script « test » sous package.json par : **"start": "nodemon index.js"**
4. Ajouter le schéma utilisateur.js suivant:

```
const mongoose = require("mongoose");

const UtilisateurSchema = mongoose.Schema({
    nom: String,
    email: String,
    mot_passe: String,
    created_at: {
        type: Date,
        default: Date.now(),
    },
});

module.exports = Utilisateur = mongoose.model("utilisateur", UtilisateurSchema);
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple (Authentification)*



5. Ajouter le fichier index.js:

```
const express = require("express");
const app = express();
const PORT = process.env.PORT_ONE || 4002;
const mongoose = require("mongoose");
const Utilisateur = require("./Utilisateur");
const jwt = require("jsonwebtoken");
const bcrypt = require('bcryptjs');

mongoose.set('strictQuery', true);
mongoose.connect(
  "mongodb://localhost/auth-service",
  {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  },
  () => {
    console.log(`Auth-Service DB Connected`);
  }
);

app.use(express.json());
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple (Authentification)*



*... Suite du fichier index.js*

```
// la méthode register permettra de créer et
d'ajouter un nouvel utilisateur à la base de
données
app.post("/auth/register", async (req, res) => {
  let { nom, email, mot_passe } = req.body;
  //On vérifie si le nouvel utilisateur est déjà
  inscrit avec la même adresse email ou pas
  const userExists = await Utilisateur.findOne({
    email });
  if (userExists) {
    return res.json({ message: "Cet utilisateur
    existe déjà" });
  } else {
    bcrypt.hash(mot_passe, 10, (err, hash) => {
      if (err) {
        return res.status(500).json({
          error: err,
        });
      } else {
        mot_passe = hash;
```

```
        const newUtilisateur = new
          Utilisateur({
            nom,
            email,
            mot_passe
          });

        newUtilisateur.save()
          .then(user =>
        res.status(201).json(user))
          .catch(error =>
        res.status(400).json({ error }));
      }
    });
  });
});
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node.js : Exemple (Authentification)*



*... Suite du fichier index.js*

```
// la méthode login permettra de retourner un token
après vérification de l'email et du mot de passe
app.post("/auth/login", async (req, res) => {
  const { email, mot_passe } = req.body;
  const utilisateur = await Utilisateur.findOne({
    email });
  if (!utilisateur) {
    return res.json({ message: "Utilisateur
introuvable" });
  } else {
    bcrypt.compare(mot_passe,
utilisateur.mot_passe).then(resultat => {
      if (!resultat) {
        return res.json({ message: "Mot de
passe incorrect" });
      }
    else {
      const payload = {
        email,
        nom: utilisateur.nom
    }
  }
});
```

```
token) => {
  jwt.sign(payload, "secret", (err,
  token ) );
};

if (err) console.log(err);
else return res.json({ token:
});
```

```
});
});

app.listen(PORT, () => {
  console.log(`Auth-Service at ${PORT}`);
});
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple (Authentification)*



Test de la méthode auth/register:

The screenshot shows a Postman test environment for a POST request to `localhost:4002/auth/register`. The request body is set to `raw` JSON, containing the following data:

```
1 {
2   "nom": "Ahmed Alaoui",
3   "email": "ahmed.alaooui@gmail.com",
4   "mot_passe": "12345"
5 }
```

The response status is `201 Created` with a duration of `198 ms` and a size of `452 B`. The response body is also JSON, showing the created user object:

```
1 {
2   "nom": "Ahmed Alaoui",
3   "email": "ahmed.alaooui@gmail.com",
4   "mot_passe": "$2a$10$0z5B6DzMHnyhzg5IGxuLCugysU8nnV0bWRfLR5y5kiI2EZfGuKMem",
5   "created_at": "2022-12-20T17:25:01.679Z",
6   "_id": "63a1f10abae7317543c96d7b",
7   "__v": 0
8 }
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple (Authentification)*



Test de la méthode auth/login:

The screenshot shows a Postman interface for testing a Node.js microservice. The request is a POST to `localhost:4002/auth/login`. The Body tab is selected, containing the following JSON payload:

```
1 {  
2   "email": "ahmed.alaoui@gmail.com",  
3   "mot_passe": "12345"  
4 }
```

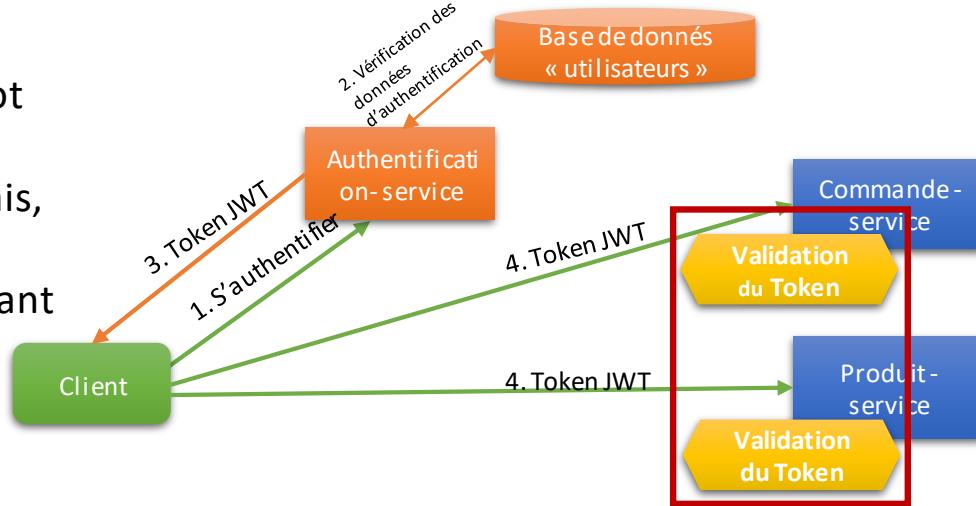
The response is a 200 OK status with a response time of 339 ms and a size of 425 B. The response body is displayed in Pretty format:

```
1 {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJlbWFpbCI6ImFobWVkLmFsYW91aUBnbWFpbC5jb20iLCJub20iOiJBaG11ZCBBbGFvdWkiLCJpYXQiOjE2NzE2NDM5NDN9.  
ObNhX0WMUhihu-Kduc0y47qCuolBsp99QFy698FMyg"  
3 }
```

## 2. Créer des microservices

### Création d'une application en microservices avec Node.js : Exemple (Authentification)

- Après avoir développé le service d'authentification et avoir généré le token suite à la connexion (email et mot de passe);
- Les services Produit et commande recevront, désormais, des tokens lors des appels de leurs APIs exposées.
- Ainsi, les dits services doivent valider le token reçu avant de répondre à n'importe quelle requête.  
 → Ajoutons à chacun des services : Produit-service et Commande-service un fichier appelé : **isAuthenticated.js**  
*dont le contenu est le suivant:*



```
const jwt = require('jsonwebtoken');

module.exports = async function
isAuthenticated(req, res, next) {
  const token =
req.headers['authorization']?.split(' ')[1];

  jwt.verify(token, process.env.JWT_SECRET,
(err, user) => {
```

```
if (err) {
  return res.status(401).json({ message:
err });
} else {
  req.user = user;
  next();
}
```

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple*



→ Modifions, par la suite, nos méthodes API, pour qu'elles prennent en considération l'intermédiaire (le middleware) **isAuthenticated**; Celui là vérifiera le token avant de répondre à la requête entrante,

Prenons, par exemple, la méthode **commande/ajouter** du service **commande-service**, son nouveau code se présentera ainsi :

```
app.post("/commande/ajouter", isAuthenticated, async (req, res, next) => {
    // Crédit d'une nouvelle commande dans la collection commande
    const { ids } = req.body;
    httpRequest(ids).then(total => {

        const newCommande = new Commande({
            produits: ids,
            email_utilisateur: req.user.email,
            prix_total: total,
        });
        newCommande.save()
            .then(commande => res.status(201).json(commande))
            .catch(error => res.status(400).json({ error }));
    });
});
```

→ L'email de l'utilisateur sera récupéré par le biais du middleware **isAuthenticated** au lieu de le passer comme paramètre de la requête POST.

## 2. Créer des microservices

### *Création d'une application en microservices avec Node js : Exemple*

Pour tester cette méthode, il faut, premièrement, démarrer les trois services, de s'authentifier et de récupérer le token et deuxièmement de d'appeler la méthode commander en lui passent ce token.

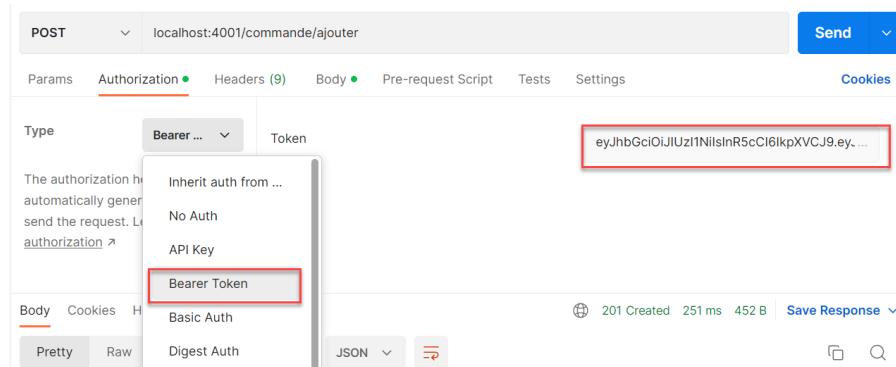
```
> produit-service@1.0.0 star
> nodemon index.js
xtensions: js,mjs,json
[nodemon] starting `node index.js`
Product-Service at 4000
Produit-Service DB Connected

> commande-service@1.0.0 start
> nodemon index.js
xtensions: js,mjs,json
[nodemon] starting `node index.js`
Commande-Service at 4001
Commande-Service DB Connected

[nodemon] watching extension
s: js,mjs,json
[nodemon] starting `node index.js`
Auth-Service at 4002
Auth-Service DB Connected
```

Le test sous Postman nécessitera deux configurations:

1. Sous « Authorization », choisir le type « Bear » et Ajouter le token récupéré après l'exécution de la méthode auth/login du service auth-service



The screenshot shows a Postman request configuration for a POST method to the URL `localhost:4001/commande/ajouter`. The 'Authorization' tab is selected, showing a dropdown menu with 'Type' set to 'Bearer ...'. A red box highlights the 'Bearer Token' option in the dropdown. To the right, the 'Token' field contains a long JWT token, also highlighted with a red box. Other tabs like 'Params', 'Headers', 'Body', and 'Tests' are visible at the top, and the bottom shows a status bar with '201 Created' and other details.

## 2. Créer des microservices

*Création d'une application en microservices  
avec Node.js : Exemple*



2. Sous Body -> row, ajouter un tableau des ids des produits à commander

The screenshot shows a Postman interface. The top bar indicates a POST request to "localhost:4001/commande/ajouter". The "Body" tab is selected, showing a JSON payload:

```
1 {
2   "ids": [
3     "639d76c1135d56ac3f30295d",
4     "639d7806cb211eb2f432b249"
5   ]
}
```

The "Response" tab shows the created command with the following JSON:

```
1 {
2   "produits": [
3     "639d76c1135d56ac3f30295d",
4     "639d7806cb211eb2f432b249"
5   ],
6   "email_utilisateur": "ahmed.alaoui@gmail.com",
7   "prix_total": 225,
8   "created_at": "2022-12-21T17:31:42.142Z",
9   "_id": "63a3433b182f8bb7bef9f48",
10  "_v": 0
11}
```

A red arrow points from the text "Commande créée" to the JSON response.

Commande  
créée

L'exécution de la requête a permis de créer la commande pour l'utilisateur authentifié.

*Le code de cet exemple est disponible sous le répertoire: <https://gitlab.com/asmae.youala/exp-microservice>*



## CHAPITRE 2

### Créer des microservices

1. Microservice versus API REST
2. Création d'une application en microservices avec Node js
3. Communication entre microservices avec Rabbitmq :
  - Principe de base de RabbitMQ ;
  - Installation du serveur Rabbitmq
  - Installation du module client amqplib avec npm ;
  - Mise en œuvre de la communication
4. Mise en place d'un reverse proxy en utilisant Nginx

## 2. Créer des microservices

*Communication entre microservices  
avec Rabbitmq*





## CHAPITRE 2

### Créer des microservices

1. Microservice versus API REST
2. Création d'une application en microservices avec Node js
3. Communication entre microservices avec Rabbitmq
4. **Mise en place d'un reverse proxy en utilisant Nginx :**
  - **Rôle d'un reverse proxy**
  - **Installation et configuration de Nginx**

## 1. Communications entre microservices

Mise en place d'un reverse proxy en utilisant  
Nginx





## Partie 4

### Manipuler les conteneurs

Dans cette partie, vous allez :

- Appréhender la notion des conteneurs
- Prendre en main Docker





# CHAPITRE 1

## Appréhender la notion des conteneurs

Ce que vous allez apprendre dans ce chapitre :

- Définition ;
- Différence entre machine virtuelle et conteneur ;
- Avantages





# CHAPITRE 1

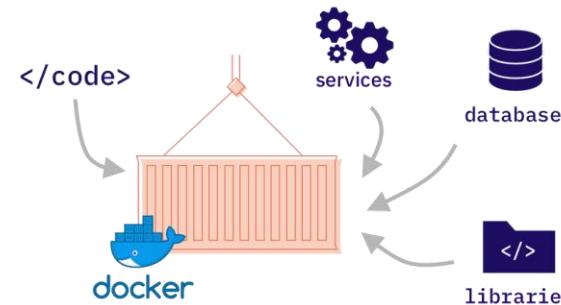
## Appréhender la notion des conteneurs

1. Définition;
2. Différence entre machine virtuelle et conteneur ;
3. Avantages

### Définition d'un conteneur



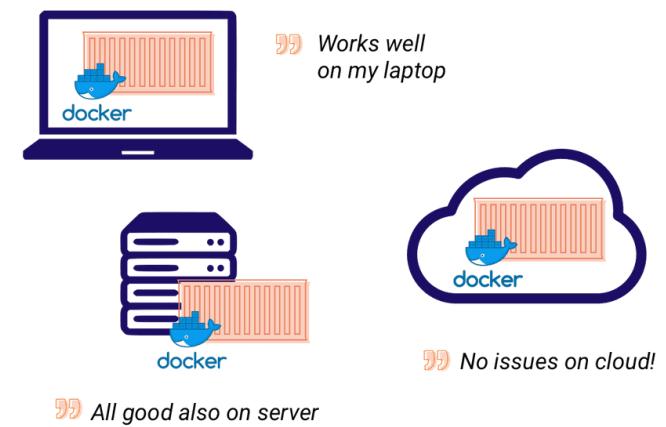
- La **conteneurisation**, est un type de virtualisation, qui consiste à rassembler le code du logiciel et tous ses composants (bibliothèques, frameworks et autres dépendances) de manière à les isoler dans leur propre « **conteneur** » ;



- Le logiciel ou l'application dans le conteneur peut ainsi être **déplacé** et **exécuté** de façon cohérente dans **tous les environnements** et sur **toutes les infrastructures**, indépendamment de leur système d'exploitation ;
- Aujourd'hui, il existe divers outils et plateformes de conteneurisation, à savoir : Docker, LXC, Podman ... ;
- **Docker** est l'écosystème le plus populaire et le plus utilisé.

### Définition de Docker

- Docker est **une plateforme de conteneurs** lancée en 2013 ayant largement contribué à la démocratisation de la conteneurisation.
- Elle permet de **créer facilement des conteneurs** et des applications basées sur les conteneurs.
- C'est **une solution open source**, sécurisée et économique.
- Initialement conçue pour Linux, Docker permet aussi la prise en charge des containers **sur Windows ou Mac** grâce à une "layer" de virtualisation Linux entre le système d'exploitation Windows / macOS et l'environnement runtime Docker.
- L'outil Docker est à la fois bénéfique pour les développeurs et pour les administrateurs système. On le retrouve souvent au cœur des processus **DevOps**.
- Les développeurs peuvent se focaliser sur leur code, sans avoir à se soucier du système sur lequel il sera exécuté. En outre, ils peuvent gagner du temps en incorporant des programmes pré-conçus pour leurs applications.





# CHAPITRE 1

## Appréhender la notion des conteneurs

1. Définition ;
2. Différence entre machine virtuelle et conteneur ;
3. Avantages

# Appréhender la notion des conteneurs

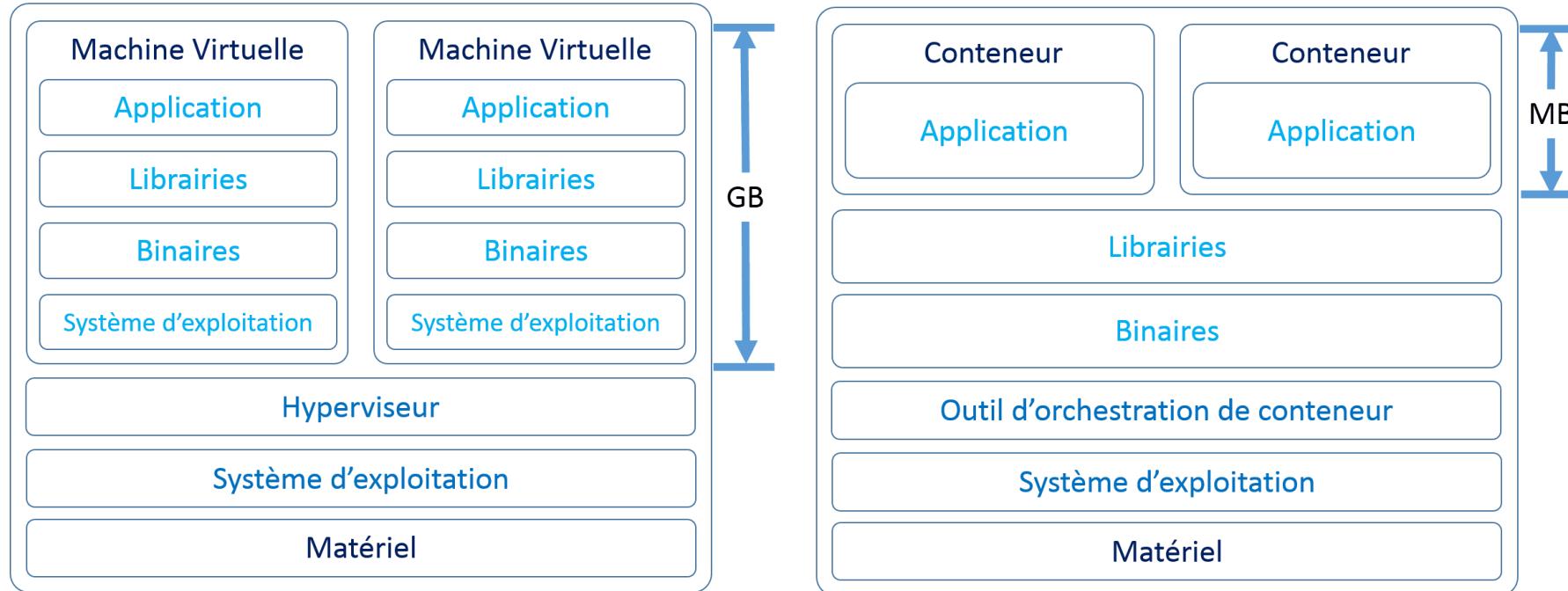
## Différence entre machine virtuelle et conteneur



	<b>Machine virtuelle</b>	<b>Conteneur</b>
<b>Performance du système</b>	Chaque machine virtuelle dispose de son propre système d'exploitation. Ainsi, lors de l'exécution d'applications intégrées à des machines virtuelles, l'utilisation de la mémoire peut <b>être supérieure</b> à ce qui est nécessaire et les machines virtuelles peuvent commencer à utiliser les ressources requises par l'hôte.	Les applications conteneurisées <b>partagent</b> un environnement de système d'exploitation (noyau), elles utilisent donc <b>moins de ressources</b> que des machines virtuelles complètes et réduisent la pression sur la mémoire de l'hôte.
<b>Légèreté</b>	Les machines virtuelles traditionnelles peuvent <b>occuper beaucoup d'espace disque</b> : elles contiennent un système d'exploitation complet et les outils associés, en plus de l'application hébergée par la machine virtuelle.	Les conteneurs sont <b>relativement légers</b> : ils ne contiennent que les bibliothèques et les outils nécessaires à l'exécution de l'application conteneurisée. Ils sont donc <b>plus compacts</b> que les machines virtuelles et <b>démarrent plus rapidement</b> .

# Appréhender la notion des conteneurs

## Différence entre machine virtuelle et conteneur



Différence entre machine virtuelle et conteneurs



# CHAPITRE 1

## Appréhender la notion des conteneurs

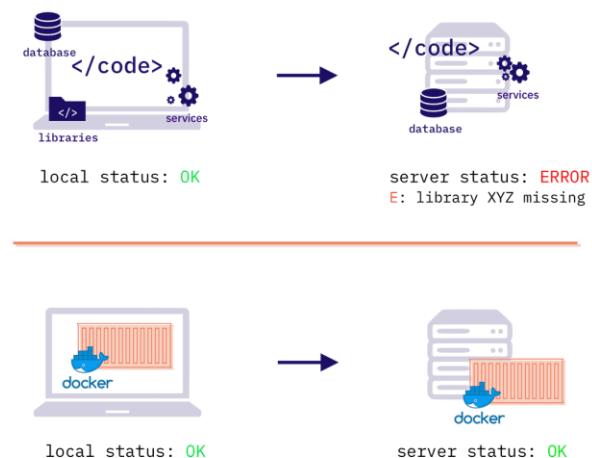
1. Définition ;
2. Différence entre machine virtuelle et conteneur ;
3. Avantages

### Avantages :

La conteneurisation offre des avantages considérables aux développeurs de logiciels et aux équipes de développement, allant d'une agilité et d'une portabilité supérieures à un contrôle des coûts amélioré. En voici un extrait de la liste :

- **Portabilité:**

Un conteneur d'application crée un progiciel exécutable qui est **isolé** par rapport au système d'exploitation hôte. Ainsi, il ne dépend pas du système d'exploitation hôte et n'est pas lié à celui-ci, ce qui le rend portable et lui permet de s'exécuter de manière cohérente et uniforme **sur n'importe quelle plate-forme ou cloud**.



### Avantages :

#### - Vitesse:

Les développeurs désignent les conteneurs comme « **légers** » parce qu'ils partagent le noyau du système d'exploitation de la machine hôte et qu'ils ne font pas l'objet de charges supplémentaires. Leur légèreté permet d'améliorer l'efficacité des serveurs et de réduire les coûts liés aux serveurs et aux licences. **Elle réduit également le temps de lancement**, car il n'y a pas de système d'exploitation à démarrer.

#### - Isolation :

La conteneurisation d'une application permet de **l'isoler** et de la faire fonctionner de façon **indépendante**. Par conséquent, la défaillance d'un conteneur n'affecte pas le fonctionnement des autres. Les équipes de développement peuvent rapidement **identifier et corriger les problèmes techniques** d'un conteneur **défectueux sans provoquer l'arrêt du reste des conteneurs**.

#### - Facilité de la gestion :

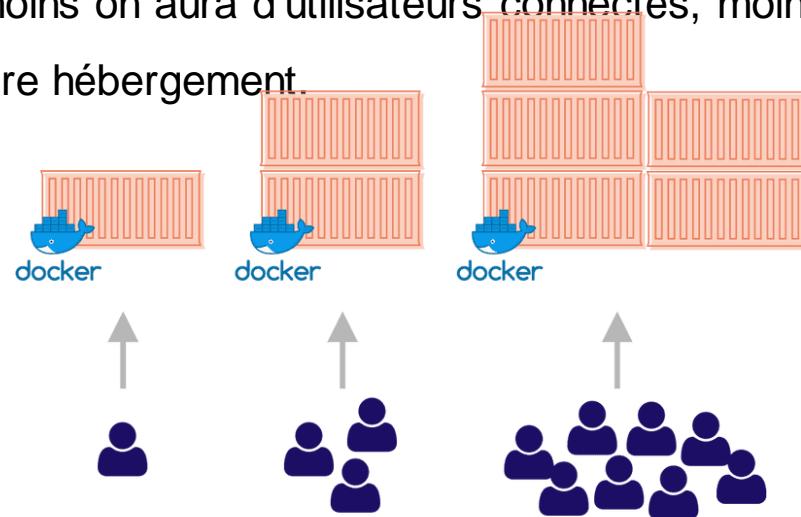
En utilisant une **plate-forme d'orchestration des conteneurs**, vous pouvez automatiser l'installation, la gestion et l'évolution des charges de travail et des services conteneurisés. L'orchestration des conteneurs permet de faciliter les tâches de gestion, comme le déploiement de nouvelles versions d'applications ou l'évolution d'applications conteneurisées ...

- **Mise à échelle flexible :**

Dans le cas d'un hébergement sur Cloud (Azure Cloud ou Google Cloud ), les conteneurs Docker peuvent être lancés en plusieurs exemplaires pour gérer le nombre croissant d'utilisateurs.

Sur le cloud, cette mise à l'échelle peut être automatisée, donc si le nombre d'utilisateurs de l'application web augmente, d'autres exemplaires de votre conteneurs seront lancés automatiquement pour pouvoir répondre à la montée de la charge.

Il en va de même pour la réduction des effectifs, moins on aura d'utilisateurs connectés, moins on aura de conteneurs démarrés, moins sera la facture de votre hébergement.





## CHAPITRE 2

### Prendre en main Docker

1. Installation de Docker Desktop ;
2. Terminologies Docker : images, Containers, Docker Hub ;
3. Création d'un Dockerfile pour une application simple node js ;
4. Introduction à Docker Compose ;
5. Gestion de plusieurs images Docker avec docker-compose :
6. Démarrage et gestion des conteneurs (commandes docker-compose) ;