Pellizzari Théo

**Prosit 1 Développement WEB : API**

## I] Mots clés

**MSA / Microservice**  
Une microservice architecture (MSA) divise une application en petits services autonomes, chacun responsable d’une fonctionnalité précise, communiquant via des API.

**Monolithique**  
Application développée comme un seul bloc contenant toutes les fonctionnalités (frontend, backend, base de données) : difficile à maintenir et à faire évoluer.

**ODM (Object Document Mapper)**  
Outil permettant de manipuler les documents d'une base NoSQL (comme MongoDB) sous forme d'objets dans un langage comme JavaScript. Exemple : Mongoose.

**API REST**  
Interface permettant la communication entre des systèmes via des routes HTTP, en respectant les principes REST (stateless, ressources, méthodes HTTP, etc.).

**API Gateway**  
Point d’entrée unique pour accéder aux microservices. Elle gère la redirection, l’authentification, la sécurité et les logs des appels API.

**MongoDB**  
Base de données NoSQL orientée documents, stockant les données en format JSON (BSON). Très utilisée pour les architectures flexibles et scalables.

**Pile MERN**  
Stack technologique composée de MongoDB, Express.js, React et Node.js. Permet de créer une application web fullstack 100 % JavaScript.

**Maturité (niveau 2 API REST)**  
Une API REST de niveau 2 utilise les **méthodes HTTP** (GET, POST, PUT, DELETE), des **URI claires** et renvoie des **codes de statut** (200, 404...).

**CRUD**  
Acronyme de Create, Read, Update, Delete : opérations de base sur les données dans une API.

**Couche d’application**  
Partie de l’application contenant la logique métier (traitements, règles de gestion). Elle fait le lien entre les données et l’interface utilisateur.

**Nginx**  
Serveur web performant utilisé comme reverse proxy ou load balancer, pour répartir les requêtes entre plusieurs services.

**Scalabilité**  
Capacité d’une application à **s’adapter à la montée en charge** (plus d’utilisateurs, plus de données) sans perte de performance.

**Orchestrateur EKS**  
Elastic Kubernetes Service d’AWS : service cloud qui facilite le déploiement et la gestion de conteneurs Docker avec Kubernetes.

**Mongoose**  
ODM pour MongoDB en Node.js. Permet de définir des schémas de données et d’interagir facilement avec la base via des modèles.

**Node.js**  
Environnement d'exécution JavaScript côté serveur. Il permet de développer des applications web performantes, non bloquantes.

**Load balancer**  
Composant répartissant les requêtes entrantes entre plusieurs instances d’un même service pour améliorer la disponibilité et la rapidité.

**Container Docker**  
Unité d’exécution légère et isolée contenant une application et ses dépendances, garantissant portabilité et reproductibilité.

**Développeur Full Stack**  
Développeur capable de travailler à la fois sur le **frontend** (interface utilisateur) et le **backend** (logique serveur, base de données).

**Portabilité**  
Capacité à faire tourner une application dans différents environnements (PC local, serveur, cloud) sans modification du code.

**Express**  
Framework minimaliste pour Node.js, facilitant la création d’API web : gestion des routes, des requêtes HTTP, middlewares, etc.

**Persistance de données**  
Fait de **conserver durablement** les données (dans une base ou un fichier) même après un redémarrage ou une panne.

**AWS (Amazon Web Services)**  
Plateforme cloud de services proposée par Amazon : serveurs, stockage, bases de données, conteneurs, IA, etc.

**Service**  
Unité fonctionnelle isolée qui fournit une ou plusieurs fonctionnalités (ex : service d’authentification, service de commandes...).

**Développement et déploiement continu**  
Pratiques DevOps pour livrer régulièrement du code fonctionnel et le mettre en production automatiquement (CI/CD).

**Application web**  
Application accessible via un navigateur, composée d’un frontend (HTML/CSS/JS) et d’un backend (API, base de données).

## II] Contexte

Yanis doit s’occuper de la migration d’une application monolithique vers une application en microservices

## III] Problématique

Comment développer un prototype d’un service backend exposant une API REST qui utilise MongoDB ?

## IV] Contraintes

- Les technologies imposées  
- L’API REST doit avoir une maturité de niveau 2  
- Architecture microservice

## V] Livrables

- Schéma d’architecture du projet  
- Microservice d’une API qui utilise MangoDB

## VI] Nature du problème

Microservice

## VII] Pistes de solutions

- Il existe des méthodes pour monter le niveau de maturité d’une API REST  
- Mangoose sert à se connecter à MangoDB depuis node js  
- Mangoose est spécifique à Node js  
- Docker permet la portabilité du microservice  
- Une API = un microservice ?  
- Service = microservice ?  
- Un microservice peut être indépendant ?  
- Utiliser un load balancer permet d’améliorer la déployabilité et la scalabilité  
- Il vaut mieux mettre le service de l’API dans un docker et la base de données de dehors  
- Est ce que l’on peut faire un modèle entité / association sur MangoDB

## VIII] Plan d’action

- Télécharger et installer MangoDB et Postman  
  
- Étudier les ressources :  
 - Microservice (Basile)   
 - API REST (Théo)  
 - MongoDB / Mangoose (Thibault)  
 - Conteneurisation Docker (Antoine)  
 - Nginx (Messaoud)  
 - Node.js / Express (Ilias)  
 - Théorème de CAP (Axel)  
- Analyse des besoins  
- Faire une comparaison entre monolithique et microservice (Matthieu)  
- Répondre aux pistes de solution (Mark)  
- Faire le schéma de l’architecture (Axel)  
- Proposer une maquette

Cours

## 1] Introduction aux Microservices – Une architecture moderne pour les applications

Aujourd’hui, les applications sont de plus en plus complexes et doivent pouvoir évoluer rapidement, être robustes et scalables. Pour répondre à ces besoins, une architecture logicielle s’est imposée : **l’architecture microservices**.

**Commençons par une comparaison**

Avant les microservices, on utilisait surtout une approche **monolithique** : toute l’application était un seul bloc. Si tu voulais modifier une partie, tu devais parfois redéployer toute l’application.

Avec les **microservices**, chaque partie de l’application devient un **service indépendant**. On découpe selon les fonctionnalités métier.

**Exemple concret :**

Une application de e-commerce peut être divisée comme ça :

* Un service pour la gestion des utilisateurs
* Un service pour les produits
* Un service pour les commandes
* Un service pour les paiements
* Un service pour l’envoi d’e-mails

Chaque service peut être développé, testé, mis à jour et déployé **indépendamment des autres**.

**Définition d’un microservice**

Un **microservice** est une **petite application autonome** qui exécute une seule tâche métier (ex : gérer les produits).

Caractéristiques d’un microservice :

* Il a son propre **code, base de données, logique**
* Il expose une **API** pour que les autres services puissent communiquer avec lui
* Il peut être **déployé seul**, sans redéployer tout le reste

**Pourquoi utiliser les microservices ?**

**Avantages :**

* **Modularité** : plus facile à comprendre, maintenir et faire évoluer
* **Scalabilité** : on peut augmenter les ressources d’un seul service (ex : les paiements) sans toucher aux autres
* **Déploiement rapide** : tu peux mettre à jour un seul service sans tout redémarrer
* **Résilience** : si un service tombe, les autres peuvent continuer à fonctionner

**Inconvénients :**

* **Complexité technique** : plus de configuration, plus de réseau, plus de débogage
* **Problèmes de communication** : les services doivent bien s’échanger les données
* **Tests plus compliqués** : il faut tester l’ensemble des interactions entre services

**Comment les microservices communiquent ?**

Il y a deux grandes façons :

**1. En synchronisé : via une API REST ou gRPC**

* Ex : un service envoie une requête HTTP à un autre service
* Rapide mais dépendant de la disponibilité de l’autre

**2. En asynchrone : via une file de messages (ex : RabbitMQ, Kafka)**

* Ex : le service "commandes" envoie un message dans une file, le service "paiements" le lit plus tard
* Plus robuste, mais plus difficile à suivre

**Un exemple d’architecture microservices**

Prenons l’exemple d’une application de gestion d’école :

* UserService : inscription, login, gestion des rôles
* CoursService : création et modification de cours
* NoteService : saisie et consultation des notes
* NotificationService : envoi d’e-mails aux élèves

Les services sont :

* **Conteneurisés** avec Docker
* **Orchestrés** avec Kubernetes ou Docker Compose
* **Connectés** via des API REST ou des messages

Un **API Gateway** est souvent utilisé pour centraliser les appels : il redirige les requêtes vers le bon service, applique des règles de sécurité, gère l’authentification.

**Chaque microservice a sa propre base de données**

Dans une architecture microservices, **chaque service a sa base de données privée**. Cela permet de garantir son autonomie.

Exemples :

* UserService → base MongoDB
* CoursService → base MySQL
* NotesService → base PostgreSQL

Il est **interdit** d'accéder à la base d’un autre service. Tout doit passer par l’API.

**Technologies utilisées**

Voici les technos les plus courantes dans ce genre d’architecture :

* **Backend** : Node.js, Java (Spring Boot), Python (FastAPI), Go
* **Base de données** : MongoDB, PostgreSQL, Redis
* **Conteneurs** : Docker
* **Orchestration** : Kubernetes, Docker Compose
* **Communication** : HTTP (REST), gRPC, RabbitMQ, Kafka
* **Authentification** : JWT, OAuth2, Keycloak
* **Monitoring** : Prometheus, Grafana, ELK (Elasticsearch, Logstash, Kibana)

**Exemple de stack Node.js avec microservices**

Un projet peut contenir :

* /auth-service/
* /product-service/
* /order-service/
* /gateway/
* /shared/ (utilitaires partagés)

Chaque dossier est une appli Node.js avec sa propre package.json, sa propre logique, sa propre base.

Ils communiquent via HTTP ou messages, et peuvent tous être **lancés dans des conteneurs** avec Docker Compose.

## II] Introduction à MongoDB – Base de données NoSQL orientée documents

MongoDB est une base de données **NoSQL** qui stocke les données sous forme de **documents JSON**, au lieu de lignes dans des tables comme dans les bases SQL. C’est une solution moderne, efficace et souple, souvent utilisée dans les applications web, les API REST, ou encore les systèmes distribués.

**Qu’est-ce qu’une base NoSQL ?**

Le terme *NoSQL* signifie "Not Only SQL". Contrairement aux bases relationnelles :

* Il n’y a **pas de schéma strict**
* Les données sont **flexibles et évolutives**
* Le stockage est souvent **horizontalement scalable** (on peut ajouter des serveurs)

MongoDB est **orienté documents** : chaque donnée est un objet JSON structuré, facile à lire, à modifier et à stocker.

**Structure d'une base MongoDB**

MongoDB fonctionne avec la hiérarchie suivante :

* **Base de données** → contient des collections
* **Collection** → contient des documents
* **Document** → objet JSON stocké (comme une "ligne" dans SQL)
* **Champs (fields)** → paires clé/valeur dans un document

**Exemple de document JSON :**

{

"nom": "Alice",

"email": "alice@mail.com",

"age": 25,

"adresse": {

"rue": "10 avenue Foch",

"ville": "Paris"

},

"langages": ["JavaScript", "Python"]

}

Ce document est stocké dans une **collection**, par exemple utilisateurs.

**Commandes de base (CRUD)**

Voici les opérations fondamentales en MongoDB :

**Créer (Create)**

db.utilisateurs.insertOne({ nom: "Alice", age: 25 });

db.utilisateurs.insertMany([{ nom: "Bob" }, { nom: "Charlie" }]);

**Lire (Read)**

db.utilisateurs.find(); // tous les documents

db.utilisateurs.find({ age: { $gt: 20 } }); // filtrer

**Mettre à jour (Update)**

db.utilisateurs.updateOne(

{ nom: "Alice" },

{ $set: { age: 26 } }

);

**Supprimer (Delete)**

db.utilisateurs.deleteOne({ nom: "Charlie" });

**Opérateurs de requête**

MongoDB permet d’écrire des requêtes très précises grâce à ses opérateurs :

* $gt, $lt, $eq, $ne : comparaisons
* $and, $or, $not : logiques
* $in, $nin : tableau
* $regex : correspondance texte

**Exemple :**

db.utilisateurs.find({

age: { $gt: 25 },

nom: { $regex: "^A" }

});

**Agrégation des données**

MongoDB dispose d’un système puissant appelé **pipeline d’agrégation**, qui permet de transformer et regrouper les données.

**Exemple simple :**

db.utilisateurs.aggregate([

{ $match: { age: { $gte: 25 } } },

{ $group: { \_id: "$age", total: { $sum: 1 } } },

{ $sort: { total: -1 } }

]);

Chaque étape du pipeline transforme les données comme dans une chaîne de traitement.

**Indexation**

Un **index** permet de rechercher plus rapidement dans une collection.

**Exemple :**

db.utilisateurs.createIndex({ nom: 1 });

MongoDB propose aussi :

* Index composés (sur plusieurs champs)
* Index texte (pour recherche plein texte)
* Index géospatiaux

**Connexion avec Node.js**

MongoDB s’intègre très bien avec JavaScript et Node.js :

npm install mongodb  
  
et

const { MongoClient } = require("mongodb");

const client = new MongoClient("mongodb://localhost:27017");

async function main() {

await client.connect();

const db = client.db("mon\_appli");

const users = db.collection("utilisateurs");

await users.insertOne({ nom: "Théo", age: 22 });

const data = await users.find().toArray();

console.log(data);

await client.close();

}

main();

MongoDB est souvent utilisé avec **Express**, **Nest.js** ou **Mongoose** (ODM).

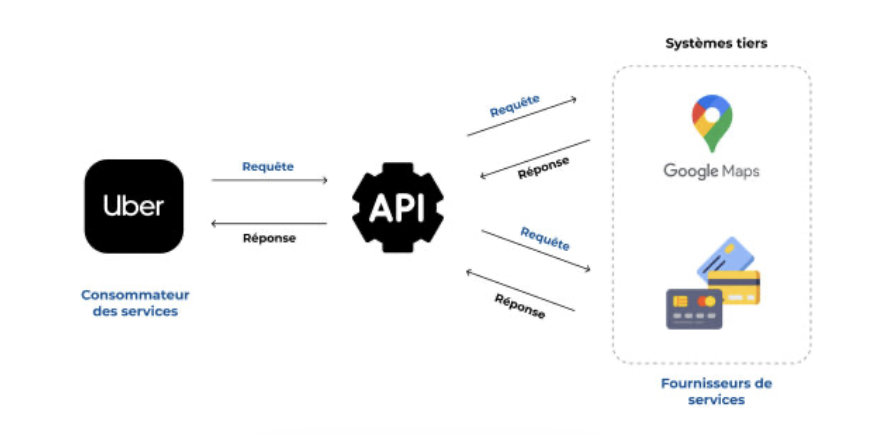
**Cas d’usage typiques**

* Appli web avec front + API Node.js
* Système de gestion de contenu (CMS)
* Messagerie en temps réel
* Données utilisateur pour un SaaS

## III] C'est quoi une API REST ?

Une **API** (**A**pplication **P**rogramming **I**nterface) est une interface de programmation permettant d’accéder à un ou plusieurs services comme des données ou des fonctionnalités fournies par un système tiers. Elle se base sur une architecture client-serveur.

Dans le schéma suivant, nous prenons l’exemple de l’application Uber (permettant de commander une course) qui fait appel à l’API de Google Maps pour avoir accès à l’itinéraire d’un client et une API de paiement pour facturer le trajet.



*Schéma d’explication du fonctionnement de l’application Uber avec API* Source : Hubvisory Source

**Rest** (**Re**presentational **S**tate **T**ransfer) est une architecture logicielle basée sur le HTTP (protocole de référence qui définit les communications sur le web), qui définit un ensemble de lignes directives architecturales à utiliser pour la création d'applications web. Les six principes REST qui guident la conception des API sont les suivants :

**1. Découplage client-serveur**

Ce principe est basé sur le fait que le client (entité qui effectue la demande des ressources) et le serveur (entité qui contient les ressources) doivent être développés de manière indépendante. Ainsi donc, en séparant les préoccupations liées à l’interface utilisateur de celles liées au stockage des données, vous améliorez la portabilité sur différentes plateformes et augmentez l’evolutivité en simplifiant les composants serveurs.

*Avantage* : le client peut évoluer d’une version à l’autre sans perturber le serveur et vice-versa.

**2. Interface uniforme**

Ce principe stipule que tout type d’appareil client devrait interagir de manière uniforme avec le serveur. Il y a quatre éléments clés à respecter pour cette contrainte:

* L’identification des ressources par un **URI (U**niform **R**essource **I**dentifier)
* La manipulation des ressources via des représentations. Quand le client envoie une demande via l’URI, il obtient une réponse (souvent en JSON) qui est la représentation de la ressource
* Les messages doivent être auto-descriptifs pour chaque demande
* Le principe **HATEOAS** ( **H**ypertext **A**s **T**he **E**ngine **O**f **A**pplication **S**tate) qui utilise des liens hypertext dans les réponses envoyées par le serveur pour permettre au client de savoir où partir chercher plus d’informations concernant la ressource

{ “products”: [ { “product-ID” : 38937, “links”:{ “rel”:”self”, “href”:”/products/38937″ } } ]}

*Avantage :* favorise la généralité, car tous les composants interagissent de la même manière

**3. Sans état (stateless)**

Ce principe stipule que le serveur ne doit pas stocker des informations relative au client(à l’exception de celles concernant l’authentification si nécessaire). Par conséquent, toutes requêtes effectuées par le client doit contenir toutes les données nécessaires pour les traiter.

*Avantage* : offre la possibilité de paralléliser les requêtes

**4. Mise en cache**

Ce principe stipule que la réponse envoyée par le serveur doit indiquer si elle est cacheable ou non et pendant combien de temps les réponses peuvent être mises en cache côté client. Si la réponse est cacheable, le client pourra récupérer cette réponse dans son propre système et il n’aura plus besoin d’envoyer de requêtes supplémentaires au serveur pour obtenir les mêmes données, ce qui améliore réellement l’optimisation de votre réseau pour les requêtes ultérieures.

*Avantage* : réduit la latence du réseau

**5. Architecture système en couches**

Ce principe indique que l’architecture doit être composée de plusieurs couches qui n’interagissent respectivement qu’avec celles voisines.

*Avantage* : Sécurité et stabilité de l’application car les composants de chaque couche ont des interactions limitées.

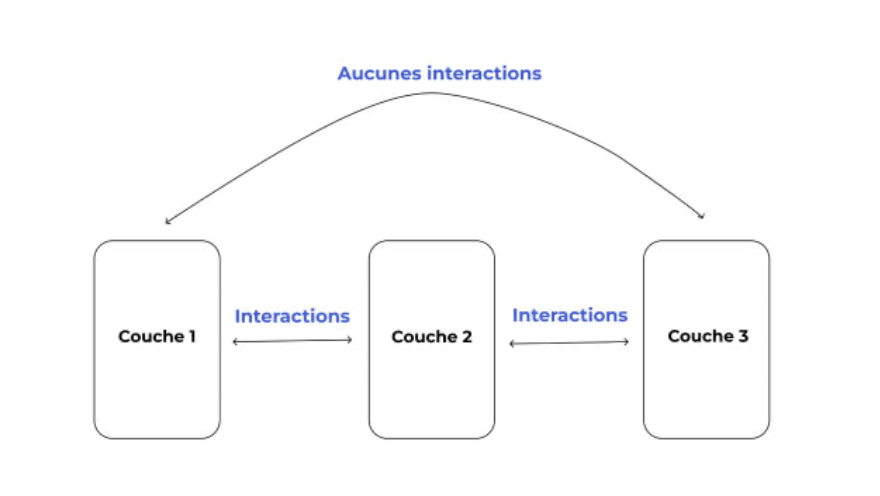
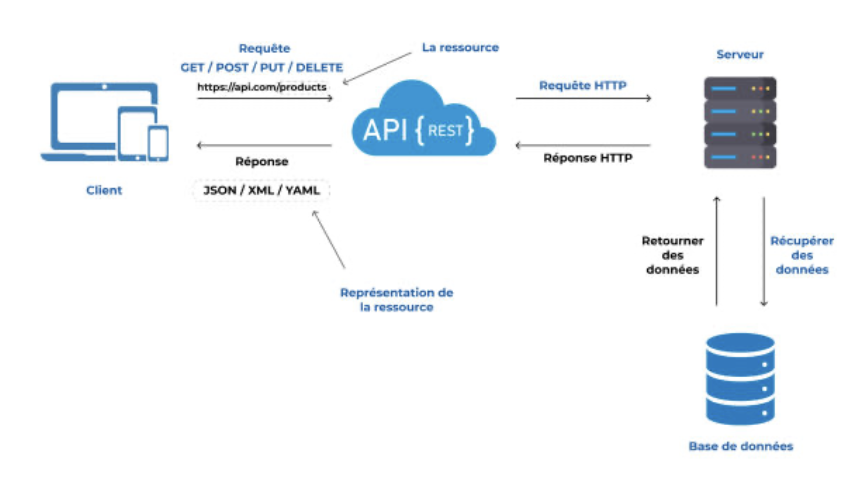


Schéma d’explication des interactions entre couches voisines.

**Comment fonctionne une API Rest ?**

Le client envoie une requête **HTTP** en précisant la ressource, le serveur traite la requête en récupérant les informations demandées dans sa base de données et ensuite renvoie une représentation de la ressource.



**Requête**

Une requête API Rest est généralement composée de :

**1. Point de terminaison**

Il contient une URI permettant au serveur de pouvoir identifier la ressource

**2. Une méthode HTTP**

Elle décrit le type de requête que le client envoie au serveur. On y retrouve les méthodes suivantes :

* **GET** récupère la représentation de la ressource
* **POST** créée de nouvelles données
* **PUT** modifie entièrement une ressource
* **PATCH** modifie partiellement une ressource
* **DELETE** supprime des données
* **OPTIONS** récupère les opérations REST disponibles**3.Header**

Les entêtes permettent de communiquer des informations utiles au client et au serveur, par exemple des données d’authentification.

**4. Body**

Le body permet de fournir des données complémentaires pour le traitement de la requête. Par exemple, des champs à modifier avec la méthode PATCH.

**Réponse**

L’API Rest répond aux différentes requêtes avec des codes réponses HTTP. Les plus courants sont les suivants :

* **200 Ok :** la demande a été acceptée et bien traitée. La ressource se trouve dans le corps de la réponse.
* **201 Created :** ressource créée avec succès. Ce code réponse est lié aux opérations de POST ou de PUT.
* **400 Bad request :** les informations renseignées dans la requête sont soit erronées soit vides, par conséquent la demande n’est pas valide.
* **401 Unauthorized :** l’utilisateur n’a pas les droits nécessaires pour accéder à la ressource demandée
* **404 Not Found :** la ressource indiquée est soit incorrecte, soit elle n’existe pas.
* **500 Internal Server Error :** problème lié au serveur ou à la demande.

Voici la [liste complète](https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) ici.

Une API REST est donc une interface de programmation d’application qui respecte les lignes directives de l’architecture logicielle Rest.

Il est important de noter qu'au delà de REST, il existe d'autres moyens de définir des services web.

**Pourquoi a-t-on besoin d'une API Rest ?**

L'utilisation d'APIs s'est démocratisée dans le domaine du développement web, son utilité principale étant d'offrir une modularité aux projets, grâce au découplage client serveur.

Il existe de nombreux format d'APIs (cf ci-dessus) mais REST continue à faire partie des formats les plus employés, et l'expression "RESTful" fait aujourd'hui partie du vocabulaire courant d'un Développeur.

Bien entendu, REST, n'est pas l'ultime solution dans le domaine du développement web et d'autres formats sont apparus pour résoudre les problématiques auxquelles REST n'était pas capable de répondre (tel que GraphQL); mais sa simplicité, sa robustesse et sa démocratisation en font un élément incontournable dans la création d'un produit numérique, que chaque Développeur, mais aussi chaque Product Owner, se doit de connaître.

## IV] Introduction à Docker et la conteneurisation

Docker est un outil qui permet de **construire, exécuter et déployer des applications dans des conteneurs**. Il est devenu un standard dans le développement moderne, surtout avec les microservices et le DevOps.

**Pourquoi utiliser Docker ?**

* **"Ça marche chez moi"** : Docker élimine les problèmes d’environnement en embarquant tout (code + dépendances).
* **Isolation** : chaque application tourne dans un conteneur séparé.
* **Léger** : plus rapide que les machines virtuelles.
* **Portable** : fonctionne partout où Docker est installé (PC, cloud, CI/CD).

**Qu’est-ce qu’un conteneur ?**

Un **conteneur** est une boîte qui contient :

* le code de ton application
* les bibliothèques nécessaires
* la configuration système

Mais **sans système d’exploitation complet**, contrairement à une VM.

* Les conteneurs partagent le noyau du système, ce qui les rend **légers et rapides**.

**Différence entre conteneur et image**

* Une **image Docker** est un **modèle figé** (template) de conteneur.
* Un **conteneur Docker** est une **instance vivante** d’une image.

L’image est comme une recette. Le conteneur est le plat préparé.

**Commandes Docker de base**

docker build -t mon-app . # Créer une image à partir d’un Dockerfile

docker images # Voir les images locales

docker run -p 3000:3000 mon-app # Lancer un conteneur

docker ps # Voir les conteneurs actifs

docker stop <id> # Stopper un conteneur

docker rm <id> # Supprimer un conteneur

**Exemple simple : application Node.js**

**Fichier Dockerfile :**

FROM node:18

WORKDIR /app

COPY . .

RUN npm install

CMD ["npm", "start"]

**Construction et exécution :**

docker build -t mon-api .

docker run -p 3000:3000 mon-api

**Docker Compose (optionnel)**

Permet de lancer plusieurs conteneurs en même temps (ex : app + base de données).

Fichier docker-compose.yml :

version: "3"

services:

web:

build: .

ports:

- "3000:3000"

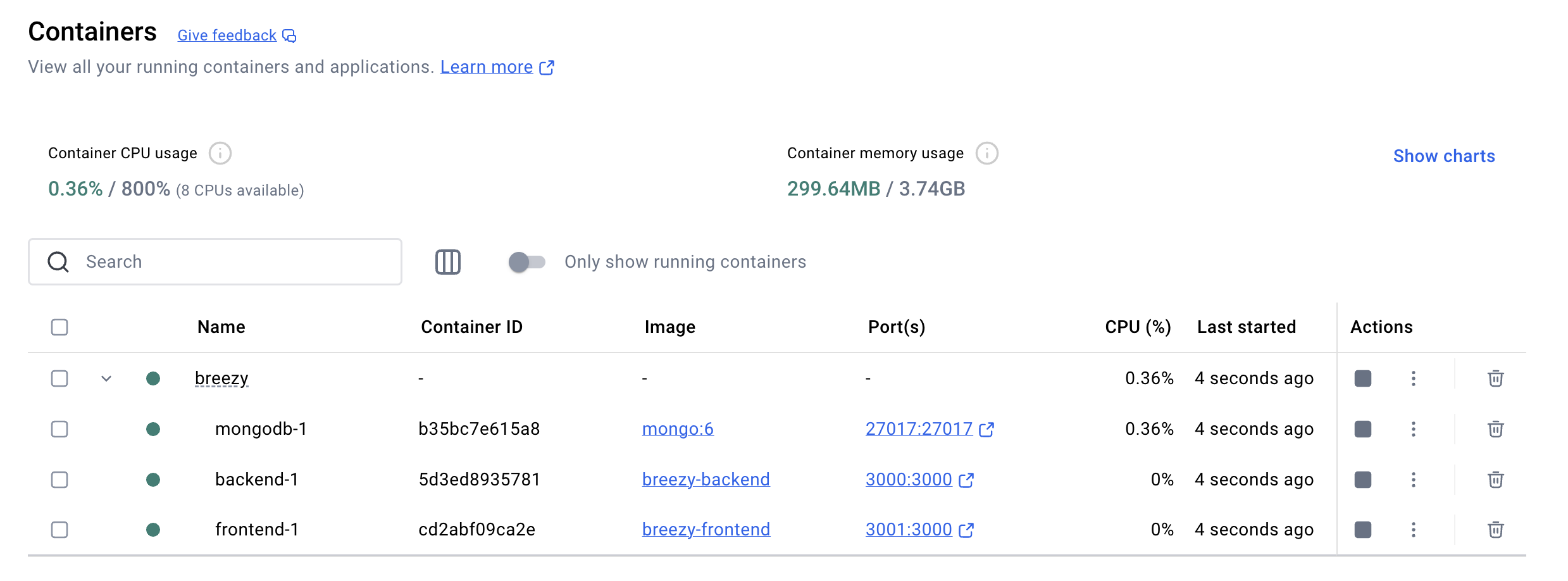
mongo:

image: mongo

ports:

- "27017:27017"

Lancez ensuite la commande docker-compose up pour créer et démarrer vos conteneurs !



## V] Introduction à Nginx – Serveur web performant et reverse proxy

**Nginx** (prononcé "Engine-X") est un **serveur web** open-source ultra léger et rapide, utilisé aussi comme **reverse proxy**, **load balancer**, et **cache HTTP**.

Il est largement utilisé dans les architectures modernes (DevOps, Docker, microservices) pour **servir des applications web, distribuer la charge, sécuriser l'accès**, etc.

**Que fait Nginx ?**

1. **Serveur web** : sert des fichiers statiques (HTML, CSS, JS, images)
2. **Reverse proxy** : reçoit les requêtes et les transmet à d'autres services (API, Node.js, Python…)
3. **Load balancer** : répartit les requêtes sur plusieurs serveurs
4. **Proxy SSL** : gère HTTPS (certificats avec Let's Encrypt)
5. **Cache** : accélère la réponse en gardant en mémoire les ressources statiques

**Cas typique d’usage**

Tu as une application backend en Node.js sur le port 3000.  
Tu ne veux pas que l’utilisateur y accède directement. Tu places **Nginx devant**, en reverse proxy :

[ Client ]

↓

[ NGINX : port 80 ]

↓

[ App Node.js : port 3000 ]

**Exemple de configuration Nginx**

Fichier nginx.conf ou fichier dans /etc/nginx/sites-available/mon-site :

server {

listen 80;

server\_name monsite.com;

location / {

proxy\_pass http://localhost:3000;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

}

}

* listen 80 : écoute sur le port HTTP
* proxy\_pass : redirige vers le backend
* proxy\_set\_header : transmet les infos client au backend

**Servir un site statique avec Nginx**

server {

listen 80;

server\_name monsite.com;

root /var/www/html;

index index.html;

location / {

try\_files $uri $uri/ =404;

}

}

**Nginx dans un conteneur Docker**

docker run -d -p 80:80 -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf nginx

Idéal pour l'utiliser en production avec une app React, Vue, ou un backend Express.

## VI] Introduction à Node.js et Express – Le duo backend JavaScript

**Node.js** et **Express** forment un environnement très utilisé pour créer des **API backend modernes**, rapides et légères, souvent utilisées avec MongoDB, React, Docker, etc.

**Qu’est-ce que Node.js ?**

**Node.js** est un environnement d’exécution JavaScript **hors navigateur**, basé sur le moteur **V8** de Google Chrome.

Il permet d’écrire du code JavaScript côté **serveur**.

**Avantages :**

* Très rapide grâce à l’event loop non bloquante
* Unifié (même langage front + back)
* Écosystème riche avec **npm**

**Qu’est-ce qu’Express ?**

**Express.js** est un **framework minimaliste** pour Node.js, qui permet de créer facilement des **API REST**, des sites web ou des middlewares.

**Pourquoi Express ?**

* Gestion simplifiée des routes (GET, POST, etc.)
* Middleware configurable
* Facile à connecter à une base de données (MongoDB, MySQL...)

**Commandes de base**

**Créer un projet Node.js :**

npm init -y

npm install express

**Lancer le serveur :**

node index.js

**Organisation typique d’un projet Express**

/mon-api

├── index.js

├── /routes

│ └── user.routes.js

├── /controllers

│ └── user.controller.js

├── /models

│ └── user.model.js

Cette architecture permet de **séparer les responsabilités** et de garder un projet clair.

**Express et MongoDB**

Avec le package mongoose, on peut connecter Express à une base MongoDB :

npm install mongoose

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/ma\_base');

III] Livrable

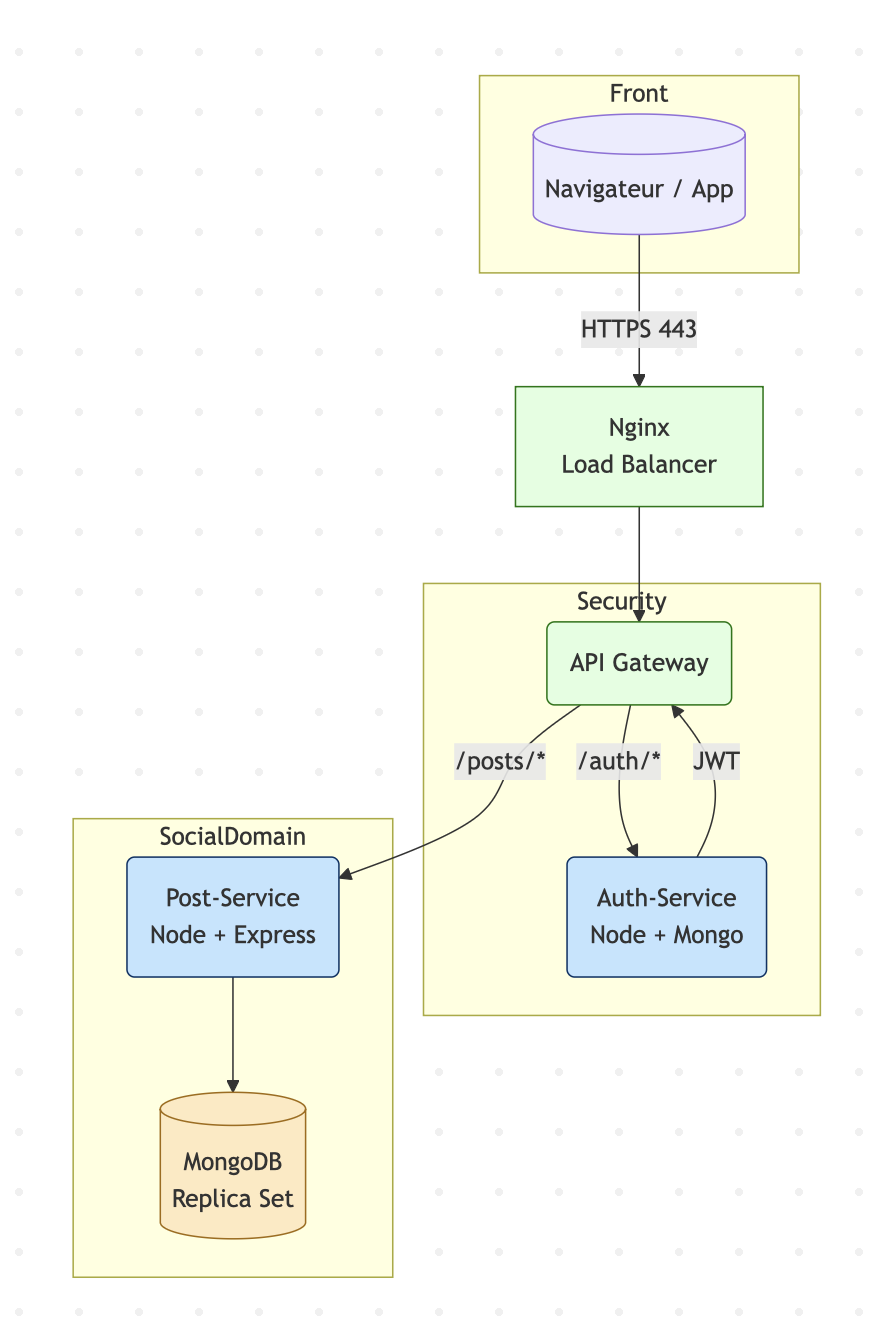
**1. Analyse des besoins**

| **Point** | **Détails** |
| --- | --- |
| **Contexte** | * Migration progressive d’une application 3-tiers monolithique vers une architecture micro-services (MSA). Le réseau social, moins critique, sert de projet pilote. |
| **Parties prenantes** | * Maia B. (architecte), Yanis (développeur MERN), équipe DevOps, lecteurs de BD (clients web & mobile), équipe produit. |
| **Fonctionnel (MVP du réseau social)** | * CRUD « posts » (texte ± images, auteur, date). * Fil d’actualité par date/auteur. * Recherche plein-texte. |
| **Non-fonctionnel** | * API **REST niveau 2** (ressources + verbes HTTP + statuts) ; JSON only * Portabilité *containers-first* : build & run identiques en local, datacenter, EKS. * Scalabilité horizontale : chaque micro-service réplicable derrière **Nginx** LB. * Sécurité : OAuth2 / JWT via *Auth-Service* + mTLS entre services. * Observabilité : logs structurés, métriques Prometheus, traces OpenTelemetry. * CI/CD : tests, scans, build d’images, déploiement automatisé (GitHub Actions → registry → Kubernetes). |
| **Contraintes** | * Stack imposée : Node.js 18 +, Express 5, MongoDB 6.x, Mongoose. * Pas de couplage entre service et base dans une même image * Pré-intégration via **API Gateway** (ex : Kong, Traefik). |

**2. Comparaison *Monolithique* vs *Micro-services***

| **Critère** | **Architecture monolithique** | **Architecture micro-services** |
| --- | --- | --- |
| **Déploiement** | 1 artefact / 1 pipeline | 1 pipeline par service |
| **Scalabilité** | verticale (augmenter la VM) | horizontale (répliquer le service concerné) |
| **Couplage** | fort (même code & DB) | faible (API + messages) |
| **Technos** | homogène | polyglotte possible |
| \*\*MTTR \* \*\* | long : redéployer tout | court : scope ≃ 1 service |
| **Tests** | simples unitaire/intégr. | tests de contrat + E2E plus coûteux |
| **Observabilité & réseau** | tracking simple | besoin tracing distribué, discovery |
| **Complexité initiale** | faible | forte (orchestration, DevOps) |
| **Quand choisir ?** | MVP, équipe réduite, peu d’exigences de charge | produit en croissance, équipes multiples, SLA élevés |

\*Mean Time To Recovery



*Shéma de l’architecture*