

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
- Mongoose sert à se connecter à MongoDB depuis node.js
- Mongoose 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 MongoDB

VIII] Plan d'action

- Télécharger et installer MongoDB et Postman
- Étudier les ressources :
 - Microservice (Basile)
 - API REST (Théo)
 - MongoDB / Mongoose (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

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] API REST

Qu'est-ce qu'une API REST ?

L'API REST est une API (Interface de Programmation d'Application) qui adhère au standard de l'architecture REST (REpresentational State Transfer). D'autres appellations désignent l'API REST : API RESTful ou API Web Restful.

Une API, ou interface de programmation d'application, est un logiciel qui permet à deux applications de communiquer entre elles. En informatique, les APIs sont indispensables pour permettre à diverses applications de fonctionner ensemble.

Conçu en 2000 par l'informaticien américain **Roy Fielding** au cours de sa thèse de doctorat, REST (*REpresentational State Transfer*) est devenu le modèle dominant de création des APIs, et un jalon incontournable dans le développement du *World Wide Web*. Aujourd'hui, la grande majorité des APIs sont basées sur REST, particulièrement pour offrir des **services Web, interactifs ou mobiles**. Découvrons le fonctionnement des API RESTful, leurs avantages, et leurs applications très variées.

Comment fonctionne une API REST ?

Dans la pratique, l'API REST fonctionne sur le principe de l'environnement client-serveur. L'API RESTful récupère et transmet d'un côté les requêtes d'un utilisateur ou d'une application, et de l'autre les informations rendues par le serveur (application ou base de données).

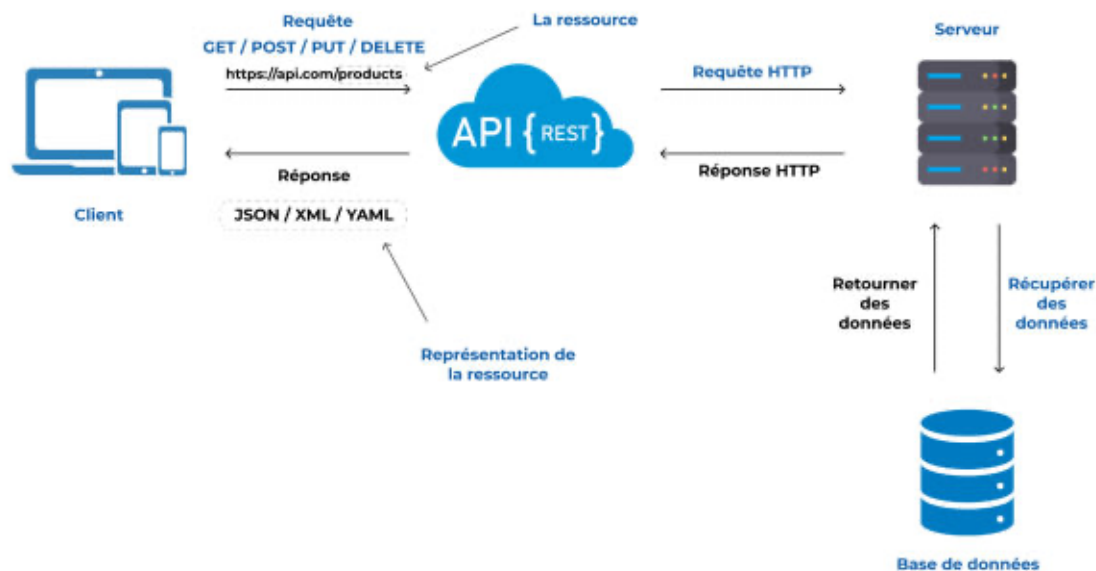
Le client est l'entité effectuant une demande. C'est le cas par exemple d'un utilisateur effectuant une recherche au sein d'un catalogue de produits sur son navigateur. **L'API** se charge de communiquer au serveur la requête, et de renvoyer vers le client les informations demandées. Les informations qui transitent par l'API sont les ressources. **Le serveur** traite les requêtes. Dans le cas présent, il renverra la liste des produits correspondant aux critères de recherche.

Les requêtes du client sont formulées à travers le **protocole HTTP** (*HyperText Transfer Protocol*). Voici les principales méthodes et tâches qu'il permet d'accomplir :

- GET : récupérer des données envoyées par le serveur.
- POST : envoyer et publier des informations vers le serveur (les données d'un formulaire d'inscription par exemple).
- PUT : mettre à jour les informations du serveur.
- PATCH : modifier partiellement une ressource existante.

- DELETE : supprimer des informations du serveur.

Les formats de données pour utiliser une API REST sont divers. Le format JSON (*JavaScript Object Notation*) est un format léger, facile à comprendre, et utilisable par de nombreux langages de programmation. XML (*Extensible Markup Language*) permet de gérer des structures de données complexes, et est compatible avec d'autres standards comme le RSS. YAML et HTML sont d'autres formats souvent utilisés pour communiquer les ressources.



Quels sont les principes de l'API REST ?

Une API REST suit les principes de REST en termes d'**architecture logicielle**. Ces principes créent une ligne directrice permettant de créer des APIs flexibles et légères, parfaitement adaptées à la transmission de données sur internet.

Voici les six principes architecturaux qui régissent une interface REST :

1. Découplage client-serveur

- **Ce que cela veut dire**
Le client (navigateur, appli mobile, script) ne connaît que l'URI de la ressource ; il ignore comment les données sont stockées, comment elles sont calculées, ou même dans quelle base elles résident.

- **Pourquoi c'est précieux**
 - **Évolutivité** : vous pouvez migrer d'un SGBD MySQL à PostgreSQL (ou à un micro-service séparé) sans changer une seule ligne côté front.
 - **Spécialisation des rôles** : l'équipe front peut se concentrer sur l'ergonomie, l'équipe back sur la performance et la logique métier.
- **Point d'attention**
Le découplage ne dispense pas d'une bonne documentation : contrat d'API clair (OpenAPI/Swagger), gestion des versions (versionnement dans l'URL ou via en-tête).

2. Interface uniforme (Uniform Interface)

- **Composants :**
 1. **Identification des ressources** (URI stables : /users/42, pas un /getUser?id=42)
 2. **Manipulation par représentation** (JSON, XML, etc. portent l'état de la ressource)
 3. **Messages auto-descriptifs** (méthodes HTTP + en-têtes expliquent la requête : PUT /users/42 = remplacement total)
 4. **HATEOAS** (liens dans les réponses : {"rel":"friends","href":"/users/42/friends"})
- **En pratique**
 - GET /articles/123 → renvoie le JSON complet.
 - Dans la réponse, un champ _links.next permet de paginer sans deviner l'URL de la page suivante.
- **Bénéfice**
Un client générique peut parcourir l'API « à la volée » sans convention privée.

3. Code à la demande (facultatif)

- **Idée**
Le serveur peut envoyer un fragment exécutable (JavaScript, WebAssembly) pour enrichir le client.

- **Cas d'usage**
 - Validations côté client transmises dynamiquement.
 - Widgets configurables (graphe, carte) téléchargés selon le besoin.
- **Limite**

À activer avec discernement : debug plus complexe, surface d'attaque plus grande.

4. Système en couches (Layered System)

- **Structure**

Client ⇌ Reverse-proxy / Gateway ⇌ Cache intermédiaire ⇌ Service d'auth ⇌ Micro-service métier ⇌ Base de données
- **Avantages**
 - **Sécurité** : un pare-feu applicatif peut filtrer.
 - **Performance** : un cache CDN sert les images sans solliciter le serveur d'origine.
 - **Flexibilité** : vous intercalez un service de limitation de débit (rate-limiting) sans toucher aux autres couches.
- **Attention**

Chaque couche doit être transparente pour le client : même URI, même contrat.

5. Mise en cache

- **Comment ça marche**

Les réponses déclarent leur « cachabilité » via les en-têtes HTTP :

 - Cache-Control: public, max-age=3600 → tout proxy peut conserver 1 h.
 - ETag: "v3.14" + If-None-Match pour la validation conditionnelle.
- **Impact**

Réduction drastique de la latence et de la charge CPU.
- **Piège classique**

Oublier d'invalidier correctement (images de profil toujours anciennes). Faites des tests en réseau simulé (Lighthouse, DevTools) pour vérifier le comportement.

6. Absence d'état (Stateless)

- **Principe**

Le serveur ne stocke *aucun* contexte de session entre deux appels ; toute information nécessaire (token JWT, préférence de langue) est portée par la requête.

- **Conséquences positives**

- **Faible encombrement mémoire** : pas de session en RAM.
- **Scalabilité horizontale** : on peut ajouter des instances de serveurs derrière un load balancer sans session sticky.

- **Écueils**

- Les URLs PUT/POST mal conçues finissent par contenir trop de paramètres.
- Pour des transactions multi-étapes, il faut une solution externe (state machine, base temporaire, saga, etc.).

Quels sont les avantages d'une API REST ?

En respectant les exigences du framework de l'API REST, les développeurs profitent des nombreux avantages des API RESTful pour développer des applications efficaces et puissantes :

- **Polyvalence** : il n'y a aucune restriction sur le langage de programmation à utiliser, et un grand choix pour le format de données (XML, PYTHON, JSON, HTML, etc.)
- **Légèreté** : les formats de données légers d'une API REST la rendent idéale pour des applications mobiles ou l'Internet des objets (IoT).
- **Portabilité** : la séparation client-serveur facilite l'échange de données entre les plateformes.
- **Flexibilité** : cette API n'a pas les lourdeurs d'un protocole, c'est un style architectural.
- **Indépendance** : les développeurs peuvent travailler séparément sur la partie client ou serveur.

Des contraintes de sécurité

La création et la gestion d'une API Web RESTful ne sont pas exemptes de défis.

L'**authentification** des utilisateurs peut devenir complexe lorsqu'elle fait appel à plusieurs méthodes différentes, par HTTP, clés API, ou OAuth (*Open Authorization*). Sur des applications larges et complexes, la **multiplication des points de terminaison** (*endpoints*) entre le serveur et le client peut nuire à la cohérence d'ensemble, de même que les mises à jour si elles laissent d'anciens points de contact encore actifs.

D'autre part, l'interface REST a une faiblesse, car elle transmet par l'URL des points de terminaison des données potentiellement sensibles, comme les identifiants. Sa sécurisation nécessite des mesures spécifiques comme un chiffrement TLS (*Transport Layer Security*), un modèle d'authentification des utilisateurs robuste, et un système de prise en charge des requêtes abusives et de limitation des débits.

Utilisations d'une API REST

Les développeurs utilisent des API à l'architecture REST pour créer et maintenir de nombreux services. Ainsi, la plupart des applications web et mobiles utilisent des API REST pour accéder et partager des ressources et des informations. Dans le Cloud, cette API permet de connecter rapidement les services des architectures distribuées et hybrides. Au sein des grandes entreprises, elle facilite l'interopérabilité entre les composants des systèmes d'information.

Voici des exemples d'application d'une API REST au sein de sociétés reconnues :

- **Google Search.** Cette API fournit en temps réel des millions de résultats de recherche sur Google Web et Google Image.
- **Deezer.** L'API REST de la plateforme de streaming musical permet de retrouver un album ou un artiste dans la base de données de millions de chansons.

Actualiser les prix d'un site e-commerce, modifier un champ dans le code source, automatiser des publications, orchestrer des conteneurs Kubernetes etc. Le champ d'utilisation des APIs RESTful n'a de limites que celle de l'imagination des développeurs et des créateurs d'applications digitales.

L'API REST de GitLab

GitLab propose une suite complète d'outils et d'APIs pour l'intégration et l'automatisation d'applications externes. Elle comprend GraphQL, les webhooks, des extensions IDE, et naturellement, une API REST. L'authentification de l'API REST GitLab peut se faire de nombreuses manières, par jeton d'accès, OAuth, ou cookies de session par exemple. Les points de terminaisons sont disponibles pour les templates Dockerfile, .gitignore, GitLab CI/CD YAML et Open source. Pour profiter pleinement de toutes les possibilités de développement de vos applications agiles et cloud-natives, consultez la [documentation complète de l'API REST de GitLab](#).

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 !

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

















Container CPU usage ⓘ

0.36% / 800% (8 CPUs available)

Container memory usage ⓘ

299.64MB / 3.74GB

[Show charts](#)

Q	Search			Only show running containers					
<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions		
<input type="checkbox"/>	 breezy	-	-	-	0.36%	4 seconds ago			
<input type="checkbox"/>	 mongodb-1	b35bc7e615a8	mongo:6	27017:27017	0.36%	4 seconds ago			
<input type="checkbox"/>	 backend-1	5d3ed8935781	breezy-backend	3000:3000	0%	4 seconds ago			
<input type="checkbox"/>	 frontend-1	cd2abf09ca2e	breezy-frontend	3001:3000	0%	4 seconds ago			

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'évent 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

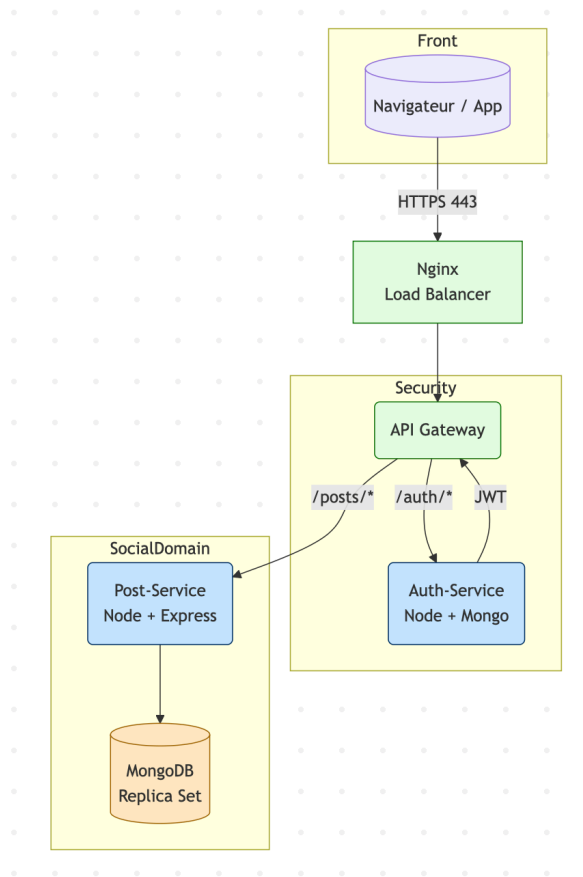
1. Analyse des besoins

Point	Détails
Contexte	<ul style="list-style-type: none">- 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	<ul style="list-style-type: none">- Maia B. (architecte), Yanis (développeur MERN), équipe DevOps, lecteurs de BD (clients web & mobile), équipe produit.
Fonctionnel (MVP du réseau social)	<ul style="list-style-type: none">- CRUD « posts » (texte ± images, auteur, date).- Fil d'actualité par date/auteur.- Recherche plein-texte.
Non-fonctionnel	<ul style="list-style-type: none">- API REST niveau 2 (ressources + verbes HTTP + statuts) ; JSON only- Portabilité <i>containers-first</i> : build & run identiques en local, datacenter, EKS.- Scalabilité horizontale : chaque micro-service répliquable derrière Nginx LB.- Sécurité : OAuth2 / JWT via <i>Auth-Service</i> + mTLS entre services.- Observabilité : logs structurés, métriques Prometheus, traces OpenTelemetry.- CI/CD : tests, scans, build d'images, déploiement automatisé (GitHub Actions → registry → Kubernetes).
Contraintes	<ul style="list-style-type: none">- 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 \approx 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

Maquette de l'application, voir : <https://github.com/theoplzr/Prosit1-Livable-DevAvanc->