## Objectif général

Concevoir et livrer une application de micro‑blogging légère, réactive et sécurisée, inspirée de Twitter/X, capable de fonctionner confortablement sur des appareils et des connexions à faibles ressources.

## Portée fonctionnelle (MVP)

Fonctionnalités obligatoires :

* **Fx1 : Création de compte** (inscription via e‑mail, validation par lien unique).
* **Fx2 : Authentification sécurisée** (login, refresh token, logout, reset MDP).
* **Fx3 : Publication de messages courts** (≤ 280 caractères, emoji pris en charge).
* **Fx4 : Affichage des messages sur le profil** (liste paginée, édition/suppression par l’auteur).
* **Fx5 : Flux chronologique** des comptes suivis (pagination infinie).
* **Fx6 : Like** (un like par utilisateur, compteur).
* **Fx7 : Commentaires sur post** (profondeur 1).
* **Fx8 : Réponse à un commentaire** (profondeur 2 max).
* **Fx9 : Suivre / Être suivi** (follow / unfollow instantané).
* **Fx10 : Profil utilisateur** (photo ≤ 2 Mo, bio ≤ 160 caractères).
* **Fx11 : Liste des messages publiés** sur profil.

## Modèle de permissions (rôles)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fonctionnalité / Rôle | Visiteur | Utilisateur | Modérateur | Administrateur |
| **Fx1. Création de comptes** | ✓ | ✗ | ✗ | ✓ |
| **Fx2. Authentification sécurisée** | ✗ | ✓ | ✓ | ✓ |
| **Fx3. Publication de messages** | ✗ | ✓ | ✓ | ✓ |
| **Fx4. Affichage des messages profil** | ✗ | ✓ *(seulement les siens)* | ✓ | ✓ |
| **Fx5. Flux chronologique** | ✗ | ✓ | ✓ | ✓ |
| **Fx6. Like** | ✗ | ✓ | ✓ | ✓ |
| **Fx7. Commenter un post** | ✗ | ✓ | ✓ | ✓ |
| **Fx8. Répondre à un commentaire** | ✗ | ✓ | ✓ | ✓ |
| **Fx9. Suivre / Être suivi** | ✗ | ✓ | ✓ | ✓ |
| **Fx10. Profil utilisateur** | ✗ | ✓ | ✓ | ✓ |
| **Fx11. Liste des messages** | ✗ | ✓ | ✓ | ✓ |

*✓ : autorisé  |  ✗ : non autorisé*

## Technologies utilisés

|  |  |
| --- | --- |
| **Domaine** | **Technologies** |
| **Back-end – Technologies** | Node.js · Express.js · JWT · MongoDB · Mongoose |
| **Sécurisation** | Authentification JWT · Gestion des erreurs · CORS |
| **Performance / Déploiement** | Docker |
| **Front-end – Technologies** | React.js · Next.js · Tailwind CSS · Axios · React Router |
| **Réactivité & UX/UI** | Mobile-first · Responsive · Gestion des erreurs UI |
| **Gestion des sessions** | Stockage des JWT · Redirection après authentification |

## Maquette de l’application

Une image contenant texte, logiciel, Icône d’ordinateur, conception

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, logiciel, Site web

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.

## Architecturechoisie

L’architecture retenue est microservices : chaque domaine fonctionnel (authentification, gestion de profil, gestion des posts) est traité par un service Node.js/Express indépendant, avec une base de données MongoDB commune. Un proxy inverse NGINX (cache/routeur) sert de point d’entrée unique, servant l’application React/Next.js et redirigeant les requêtes vers les API correspondantes. Cette approche, où « l’application est composée de services indépendants » communicant via des API bien définies, correspond à la définition d’une architecture microservices.

Une image contenant capture d’écran, texte, diagramme, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Figure 1 Architecture microservice du projet

La base de données est organisée sous MongoDB en collections correspondant aux modèles de données (User, Auth\_User, Profile, Post). Le schéma NoSQL est simple : les documents utilisateur et authentification sont liés par userId, chaque profil contient les listes d’abonnés/abonnements, et chaque post référence son auteur ainsi que les likes et les réponses associées. On peut schématiser ce modèle relationnel en MongoDB comme suit, d’après le diagramme fourni (modèle de données) :

Une image contenant texte, capture d’écran, diagramme, nombre

Le contenu généré par l’IA peut être incorrect.

Figure 2 Modèle de données MongoDB : collections Auth\_User, User, Profile et Post liées via des références d’\_id. Par exemple, Auth\_User.userId pointe vers User.\_id.

En pratique, les variables d’environnement définissent le même URI Mongo (mongodb://mongo:27017/bddTest) pour tous les services, qui partagent ainsi le même cluster. Chaque service expose son API à un port spécifique (3000 pour l’auth-service, 3001 pour profile-service, 3003 pour post-service) et est configuré dans NGINX pour que les chemins /auth-service, /profile-service, /post-service soient redirigés vers le conteneur adéquat. Le client Next.js tourne sur le port 3000 dans son conteneur (exposé sur 3001) et est servi sous le chemin racine. Les conteneurs communiquent sur le réseau Docker « my-network » défini dans le docker-compose.yml. L’utilisation de Docker Compose facilite le déploiement de cette architecture complète (services et DB) en un seul réseau virtuel.

## Hiérarchisation des taches et fonctionnalités

La planification a été guidée par les priorités du MVP : les tâches fondamentales (création de compte Fx1 et connexion Fx2) ont été réalisées en premier, permettant de sécuriser l’accès à l’application. Ensuite, les fonctionnalités de base du microblogging ont été abordées : création et affichage de posts (Fx3, Fx4, Fx11) via l’authentification établie. Puis les interactions sociales ont été implémentées : le flux des posts suivis (Fx5) combinant les services « follow » et publication, le système de likes par post (Fx6), et enfin le module de commentaires et réponses (Fx7–Fx8). Les fonctionnalités de suivi (« follow »/« unfollow », Fx9) et la page profil de l’utilisateur (Fx10) ont également été intégrées pour compléter le périmètre. Cette progression du simple vers le complexe garantit que chaque couche de fonctionnalités repose sur la précédente, conformément aux pratiques agiles d’établissement du MVP (méthode Scrum/Kanban). Par exemple, le service de suivi a nécessité que les profils existent, et le fil d’actualité dépend des relations de « follow ».

En résumé, l’équipe a d’abord implémenté les user stories liées à l’authentification et la création de profils, puis la gestion des posts, puis les interactions sociales, de sorte que chaque étape fournisse un produit utilisable et testable avant d’ajouter les suivantes. Les exigences non techniques (maquettes, modèles de données, déploiement Docker) ont été traitées en parallèle pour soutenir le développement des fonctionnalités métier.

## Methodologie suivie et étapes

L’équipe a utilisé Git pour le contrôle de version (commit régulier, probablement en branches de fonctionnalités) et un dépôt GitHub pour le partage. Chaque membre a pu travailler sur un microservice séparé grâce à cette architecture. La maquette fonctionnelle dans le front-end a été testée directement via le navigateur, et chaque service a été testé au moyen d’outils comme Postman et MangoDB.

Concernant le cycle de développement, l’équipe a commencé par établir le modèle de données et les routes principales (authentification, gestion des profils, gestion des posts). d’abord l’inscription et la connexion, puis la création de posts et l’affichage du fil, puis les interactions (like/commentaires) et enfin la finition du profil utilisateur. De plus, l’utilisation de Docker Compose a servi à vérifier l’intégration globale : à chaque modification, on rebuildait les images (docker-compose build) et relançait l’ensemble pour tester les communications entre conteneurs.

Enfin, le déploiement est géré par les Dockerfile de chaque service et par le fichier compose. Les conteneurs ont été exécutés localement (via Docker Desktop).

## Étapes et ressources mobilisées

Le back-end est développé en Node.js avec Express pour chaque microservice. Les dépendances principales incluent *Express.js* pour le serveur web, *Mongoose* pour l’ORM MongoDB, *bcryptjs* pour le hachage de mots de passe, *jsonwebtoken* pour la génération de JWT, et *dotenv* pour la gestion de la configuration via .env. Les modèles (models/) définissent la structure des données, les contrôleurs (controllers/) gèrent la logique métier et les routes (routes/) exposent les API REST. Chaque service suit une architecture MVC basique : par exemple, lors de l’inscription d’un utilisateur, le auth-service crée un document User, un document Auth\_User (contenant email et hash du mot de passe) et un document Profile initial, comme vu dans le code. Les ports et connexions DB sont configurés via des variables d’environnement (fichiers .env). Côté base de données, MongoDB est utilisé en mode conteneur unique, et tous les services y accèdent via Mongoose.

Tech stack (front-end). Le client web est une application React avec Next.js, permettant le rendu côté serveur (SSR) ou client. Le layout est défini dans src/app/layout.js, et les composants incluent une barre de navigation (Navbar.js) avec des icônes React (AiOutlineHome, etc.), une carte de publication (PostCard.js), un spinner de chargement (Loader.js), etc. Pour le style, Tailwind CSS est utilisé (via tailwind.config.js et postcss.config.mjs). La communication avec les services s’effectue via Axios à partir d’URL relatives (le proxy NGINX active le routage), et la gestion de l’authentification se fait via un contexte React (AuthContext.js) qui stocke le JWT (dans un cookie HttpOnly) et les infos utilisateur. L’application est conçue responsive en mobile-first, avec un fil d’actualité présent à l’accueil (page.js) et des pages dédiées pour le profil utilisateur.

Le dépôt Git est structuré en dossiers par service : auth-service, profile-service, post-service, et client pour le front-end. Chaque dossier de service contient son propre package.json, server.js (point d’entrée Express) et dossiers models, routes et controllers. Le reverse-proxy NGINX a un dossier nginx avec Dockerfile et nginx.conf. La compilation et le déploiement sont orchestrés par un docker-compose.yml à la racine, qui définit un réseau my-network et les conteneurs pour MongoDB, les trois services, le client Next.js et le proxy NGINX. Cette organisation modulaire permet de déployer chaque composant indépendamment et de mettre à l’échelle les services critiques séparément si besoin (par exemple répliquer le service post ou profil).

En résumé, l’équipe a mobilisé les ressources suivantes : Node.js/Express et MongoDB/Mongoose pour les microservices, React/Next.js/Tailwind pour le front-end, NGINX pour le proxy, et Docker/Docker Compose pour l’intégration et le déploiement containerisé.

## **Interfaces utilisateurs (maquettes/**wireframes**)**

Le frontend propose plusieurs écrans clés : une page d’accueil (feed) listant les posts des utilisateurs suivis, une page de connexion/inscription pour les nouveaux utilisateurs, et une page profil affichant les informations de l’utilisateur et ses propres posts. L’interface comprend une barre de navigation en haut (composant Navbar.js) avec des liens/icônes vers l’accueil, les notifications (non implémentées), et le profil. Les posts sont affichés sous forme de cartes (PostCard.js) montrant l’auteur, la date, le contenu, et des boutons pour « like » et « commenter ».

Par exemple, sur la page de connexion, un simple formulaire React avec champs email/mot de passe est soumis à l’API /auth-service/login. Sur la page d’accueil, on trouve un champ de création de nouveau post (via un textarea géré en React) et en dessous la liste des posts (récupérés de /post-service/posts/following). Le design est responsive grâce à Tailwind : en mobile, la navigation devient un menu hamburger et les cartes de post s’ajustent en largeur. Les captures d’écran fournies semblent montrer un backend Docker (pas d’UI), donc nous présentons ici un schéma conceptuel inspiré du code existant. Par exemple, PostCard contient le nom de l’auteur (via populate en back-end), un bouton pour aimer qui déclenche toggleLike, et un bouton pour répondre qui ouvre un mini-formulaire. Les erreurs (champs vides, échec d’API) sont gérées par des messages basiques à l’utilisateur.

## **Fonctionnalités principales et leur implémentation**

**Inscription (Fx1).** Le service auth-service offre la route POST /auth-service/register. Le contrôleur vérifie d’abord qu’aucun Auth\_User n’existe déjà avec le même email, et qu’aucun User n’a le même nom d’utilisateur. Il hache ensuite le mot de passe avec bcrypt (bcrypt.hash(password, 10)) avant de sauvegarder les trois documents : un nouveau User (stockant username, email), un Auth\_User (avec userId, email, passwordHash, rôle), et un Profile vierge associé (avec displayName = username). Si l’unique lien de validation par email était implémenté, ce serait ici que l’on générerait un token et enverrait un mail. En l’état, l’utilisateur est créé actif immédiatement.

**Connexion et JWT (Fx2).** La route POST /auth-service/login reçoit l’email et le mot de passe. Après avoir retrouvé le document Auth\_User correspondant en base, le mot de passe est comparé (bcrypt.compare). Si valide, le service génère un JWT signé avec une clé secrète (dans le code, « jwttokenkey ») qui contient le payload {userId, roles, username}. Le JWT est ensuite envoyé dans un cookie HttpOnly et renvoyé aussi dans la réponse JSON. Ce token rend l’authentification **stateless**: le serveur n’a pas besoin de sessions serveur et se fie au token pour vérifier l’utilisateur sur chaque requête. (Les permissions d’accès aux routes futures utiliseraient ce token dans les headers Authorization.) Un cookie de rafraîchissement pourrait être ajouté pour prolonger la session, mais n’est pas implémenté ici. Un logout pourrait consister à supprimer ce cookie.

**Publication de messages courts (Fx3).** Le service post-service expose POST /post-service/posts pour créer un post. Le contrôleur createPost reçoit userId, content (et optionnellement media si image/vidéo). Il vérifie que l’utilisateur existe, puis instancie un Post avec le contenu haché et le tableau vide pour likes et replies. Le post est sauvegardé et renvoyé au front, parfois avec les champs peuplés. La taille des messages n’est pas explicitement limitée, mais Next.js limite les uploads via <textarea> (cependant, un contrôle côté front-end ou schéma (maxlength) pourrait être ajouté). Dans le fil d’actualité (GET /post-service/posts ou /posts/following), l’API renvoie les posts triés par date décroissante. Les données retournées incluent le nom d’utilisateur de l’auteur grâce à .populate("userId", "username displayName").

**Flux chronologique (Fx5).** Pour afficher le fil des comptes suivis, le front demande à l’API POST /post-service/posts/following en envoyant un tableau followingIds d’IDs d’utilisateurs suivis (obtenus depuis le profil de l’utilisateur courant). Le contrôleur getPostsByFollowing fait une requête Mongo find({ userId: { $in: followingIds } }) triée par date. Les posts des utilisateurs suivis sont alors renvoyés au front, qui les affiche sous forme de liste infinie (scroll infini). Cette page est accessible uniquement après login, via le token JWT.

**Like (Fx6).** La route POST /post-service/posts/:postId/like implémente la fonction de like. Elle reçoit le userId (dans le corps) et cherche le post. Si le userId n’est pas dans la liste post.likes, il est ajouté ; sinon il est retiré (toggle). On sauvegarde le post mis à jour, puis on renvoie le post « peuplé » au front (avec .populate comme précédemment). Le front met à jour l’affichage (le compteur de likes est la longueur de post.likes). Cette logique permet un like unique par utilisateur.

**Commentaires et réponses (Fx7–Fx8).** Les posts contiennent un tableau de replies de profondeur 1. La route POST /post-service/posts/:postId/replies ajoute une réponse. Le corps doit contenir userId et content. Le contrôleur trouve le post, pousse un objet {userId, content} dans post.replies, puis renvoie le post peuplé. Le front affiche alors le commentaire sous le post. Ce mécanisme correspond aux commentaires de profondeur 1 (Fx7). La « réponse à un commentaire (Fx8) » n’est pas gérée nativement car la structure enregistre seulement les réponses au post, pas aux commentaires. Il aurait fallu un schéma plus complexe (ou un second schéma Replies imbriqué), non implémenté ici.

**Suivre / être suivi (Fx9).** Dans le profile-service, la route POST /profile-service/follow gère l’abonnement. Elle reçoit dans le corps currentUserId et userIdToFollow. Après vérification des deux profils, on utilise Mongoose pour basculer l’abonnement : on vérifie si userIdToFollow est déjà dans currentUserProfile.following; si oui, on supprime userIdToFollow de following et currentUserId de targetUserProfile.followers (unfollow), sinon on ajoute l’un et l’autre (follow). L’API renvoie la nouvelle valeur booléenne following: true/false pour actualiser l’interface (changement du label bouton). Cette méthode manipule directement les tableaux de références ObjectId du profil, puis sauvegarde les deux documents.

**Profil utilisateur (Fx10, Fx11).** La gestion du profil est assurée par plusieurs routes du profile-service : création de profil (POST /profile-service), lecture (GET /profile-service/:userId), mise à jour (PUT /profile-service/:userId), suppression (DELETE). Les champs incluent displayName, bio, profileImage, etc. Actuellement, les images de profil et de bannière sont supposées être des URL (pas d’upload de fichier implémenté). La longueur du bio n’est pas vérifiée (on pourrait limiter à 160 caractères). La liste des posts sur le profil d’un utilisateur est obtenue par GET /post-service/posts/user/:userId. Le back-end renvoie tous les posts dont userId correspond, ce qui permet d’afficher l’historique des publications de l’utilisateur.

En résumé, chaque fonctionnalité principale correspond à une route/API dans le code : les opérations sur les utilisateurs (inscription, login) sont dans auth-service, les opérations sociales dans post-service et profile-service. Ces services exécutent chacun leur logique indépendante sans logique métier partagée, illustrant l’architecture découplée. L’usage d’un token JWT pour l’authentification illustre le principe de session sans état (stateless), évitant le besoin de stocker les sessions côté serveur.