Supinfo International University

3PROJ

Projet de fin d'année

Kamul ALI NASSOMA WATTARA

Corentin HOAREAU

Table des matières

Introduction

Pour notre projet de fin d'année de Beng3 à Supinfo-Paris, nous avons eu comme tâche de mettre en place une plateforme d'hébergement vidéo, comprenant infrastructure, backend, et frontend. L'infrastructure doit être sécurisée et résiliente.

Ce document technique détaillera les différentes phases du projet.

Il débutera par une phase de conception, durant laquelle notre groupe analysera le sujet pour élaborer un plan de travail.

Suite à cela, le document détaillera la mise en place de l'infrastructure prévue, ainsi que les solutions qui ont pu être abandonnées ou changées durant la phase de réalisation du projet. Cette section abordera les différents aspects du projet comme l'écriture du code, la configuration Ops, etc.

Enfin, notre groupe pourra faire un bilan de ce qui a pu ressortir au cours de ce projet.

Conception

Cette partie détaille dans les grandes lignes les différents aspects de notre infrastructure.

Frontend

Le Front est développé localement avec React, son déploiement nécessite moins de configuration que le backend, il n'a donc pas besoin d'être dockerisé lors de l'étape de développement.

Backend

Le backend de notre application est réalisé avec NestJS.

La qualité du code est assurée par Typescript, apportant du typage fort à Javascript, Eslint, et Prettier.

La base de données utilisée est MongoDB Server, cette BDD propose plusieurs avantages :

- Une forte intégration avec NestJS, grâce à la couche de compatibilité @nestjs/mongoose ;
- De meilleurs performances comparé à une base de données relationnelle, au coût de relations plus simples ;
- Une syntaxe basée sur du JSON/Javascript, facilitant la création de requêtes complexes comparé au SQL.

Les mots de passes sont tous chiffrés en base de données, et mongoose offre une protection native contre les attaques les plus communes (Injection NoSQL).

Docker

Le système de conteneurisation Docker permet de déployer facilement des conteneurs isolés, grâce à cela, les configurations serveur seront moins affectées par la machine du développeur ou l'environnement du serveur final. Cela facilite aussi le développement collaboratif.

Kubernetes

Kubernetes est utilisé pour assurer la haute disponibilité de notre infrastructure. Nous avons choisi cette solution car elle s'intègre bien à docker, peut fonctionner sur une seule machine, et est moins lourd à la mise en place qu'un groupe de machines virtuelles. Nous sommes convaincus que la conteneurisation est l'une des meilleures options pour répondre à des projets demandant une infrastructure serveur complexe.

Nginx

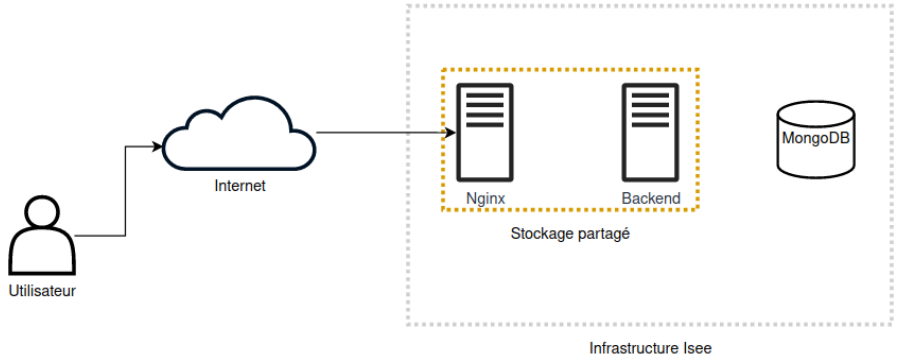
Un serveur Nginx est utilisé dans ce projet en tant que "Reverse Proxy", il s'agit du seul serveur qui est exposé à l'extérieur, et qui récupère toutes les requêtes avant de les rediriger vers notre serveur backend, ou bien de les traiter lui-même lorsqu'il s'agit de fournir des fichiers statiques.

Stockage des fichiers

Les vidéos, leurs miniatures, ainsi que les photos de profile, doivent être accessibles par le serveur Nginx en lecture seule pour être délivrés à l'utilisateur, mais ils doivent aussi être accessibles par le backend pour effectuer des actions telles que le traitement des vidéos et la création des miniatures. Par conséquent, un système de stockage partagé doit être mis en place.

Infrastructure

Le schéma de notre infrastructure est donc le suivant.



Il s'agit d'un schéma simplifié ne représentant pas l'aspect de conteneurisation ou de mise à l'échelle, et n'explicitant pas la solution choisie pour le stockage partagé.

Base de données appli web ISee

Enumerations

EUserRole : user, admin

EVideoState : draft, public, unlisted, private, deleted, blocked, uploader_deleted

EVideoProcessing : not_started, in_progress, done, failed

Subdocuments

Les subdocuments ne sont pas stockés en base de données, leur rôle est de s'imbriquer dans les schémas.

UserState
isEmailValidated : bool
isDeleted : bool
isBanned : bool

ReducedUser
_id: ObjectId
username : string
avatar : string

Schémas

Les schémas sont stockés en base de données

User
_id : ObjectId
username : string
bio : string
password : string
email : string
avatar : string
state : UserState
role : EUserRole
likedComments: string[]
likedVideos: string[]
createdAt: Date
updatedAt: Date

Video
_id : ObjectId
title : string
description : string
thumbnail : string
videoPath: string
state: EVideoState
views: number
size: number
likes: number
uploaderInfos: ReducedUser
processing: EVideoProcessing
createdAt: Date
updatedAt: Date

Comment
_id: ObjectId
content: string
(indexé) videoId: ObjectId
isEdited: bool
authorInfos: IReducedUser
likes: number
createdAt: Date
updatedAt: Date

Planification

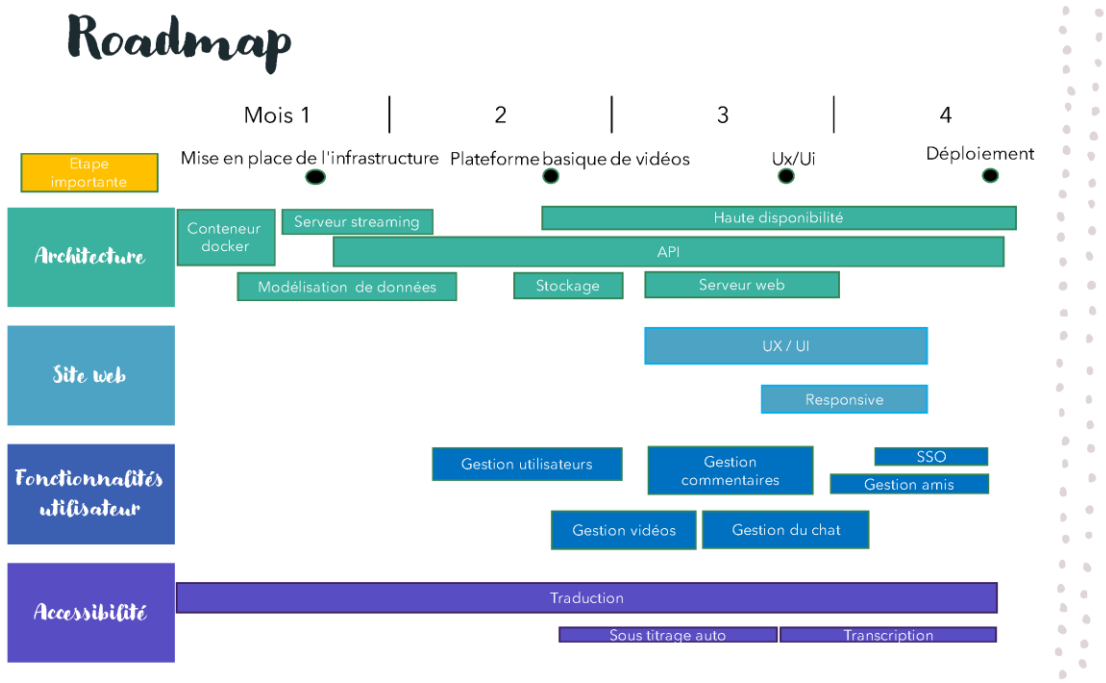
Attribution des tâches

Les tâches ont été réparties de la manière suivante

Kamul ALINASSOMA WATTARA	Corentin HOAREAU
Planification	Conception données
Maquettage	FrontEnd
Kubernetes, Backend	

Roadmap

Les étapes du projet ont été définies comme suit :



Wireframe du site

Nous avons défini la structure du site basé sur ce prototype en mettant en avant les fonctionnalités et la facilité d'accès.

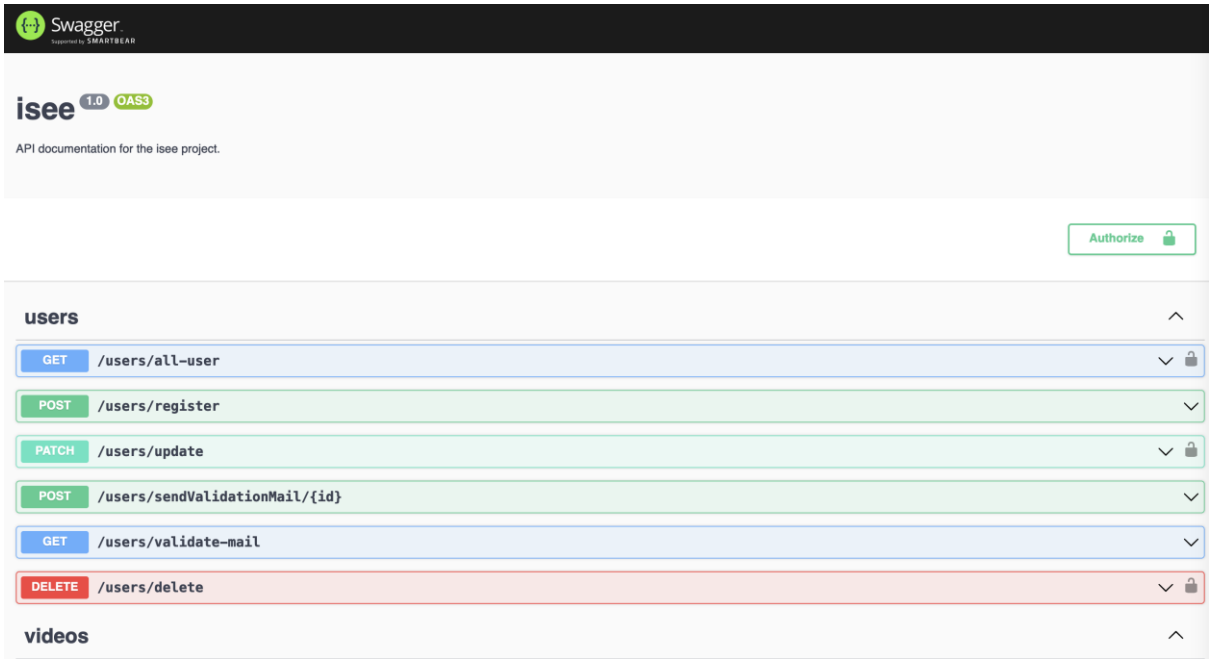
- La page d'accueil (annexe : wireframe/accueil) est principalement orientée vers les utilisateurs finaux, et affichera le fil d'abonnement de l'utilisateur ainsi que les vidéos recommandées
- La page d'une vidéo (annexe : wireframe/watch-video)permet de lire celle-ci, voir les vidéos recommendées par rapport à la vidéo actuelle, commenter, et accéder au chat
- Les pages utilisateurs
    - Une page listant l'activité de l'utilisateur (annexe : wireframe/account-videos)
    - Une page donnant des informations additionnelles sur un utilisateur (annexe : wireframe/account-infos)
    - Une page de gestion administrateur (annexe : wireframe/account-admin-dashboard)

Documentation

L'API est documentée grâce à Swagger, qui s'intègre avec notre backend NestJS

Cette documentation technique n'est accessible que sur un environnement de développement, et doit être désactivée lors du déploiement public du backend.

Cette page, accessible depuis l'URL /docs, recense tous les Endpoints exposés au public, ces Endpoints seront appelés par le frontend



Une documentation additionnelle existe dans le code, La syntaxe JSDoc est utilisée afin de fournir des détails plus techniques sur certaines fonctions.

```
/**
 * Finds a single video by mongo ID.
 * @param id
 * @returns
 */
async findOneById(id: string) {
  return await this.videoModel.findById(id);
}

/**
 * Deletes a single video by mongo ID.
 * @param id
 * @returns
 */
async deleteVideoById(id: string) {
  const video = await this.videoModel.findByIdAndDelete(id);
  return video;
}
```

Création

Cette partie donne plus de détails sur la création du projet, les problèmes rencontrés, et les solutions utilisées.

Mise en place du Backend

Le backend est l'élément le plus important de ce projet. Il était donc essentiel qu'il soit suffisamment robuste. Notre groupe a donc fait le choix d'utiliser le Framework NestJS pour sa mise en place.

NestJS facilite la création d'un serveur backend grâce à ses fonctionnalités :

- Un système de Modules permettant l'injection de dépendances dans toute l'application ;
- Un fonctionnement Service/Controller permettant de distinguer le code métier et la logique API ;
- Des Guards pour assurer des fonctions de validation ou autorisation avant d'accéder au code des contrôleurs ;
- Des Décorateurs pour ajouter de la fonctionnalité additionnelle réutilisable ;
- Des Intercepteurs pour permettre une gestion d'erreurs et une Journalisation plus avancée.

En plus de cela, NestJS propose des modules additionnels tels que l'intégration de Mongoose, permettant d'utiliser une approche par décorateurs afin de déclarer nos schémas.

Serveur NGINX

Le serveur NGINX a plusieurs rôles dans notre infrastructure, il sert à la fois :

- De reverse proxy, c'est donc la « porte d'entrée » à notre serveur backend ;
    - Les fichiers, vidéos, et miniatures de vidéo sont servies directement par le serveur Nginx ;
    - Tout le reste des requêtes est relayé au serveur Backend ;
- De serveur de streaming HLS, permettant une lecture fluide des vidéos, avec une la possibilité d'ajuster automatiquement la qualité.

Ainsi, l'utilisation de NGINX nous aide pour la réalisation d'un backend plus modulable et résilient. Il est alors possible de rajouter des nouveaux serveurs backend de manière transparente pour le frontend, car le serveur contacté restera dans tous les cas le serveur Nginx.

Relations et mises à jour

Un problème que nous avons identifié très tôt dans la phase de planification est celui des relations entre un utilisateur et ses vidéos/commentaires.

En effet, ses éléments doivent afficher des informations concernant leur auteur, principalement :

- Son id MongoDB ;
- Son nom d'utilisateur ;
- Un lien vers son avatar.

Il est peu coûteux de récupérer ces informations lorsque l'on charge une seule vidéo, mais cela peut s'avérer plus problématique lorsqu'on charge une liste de vidéos ou de commentaires : bien que Mongoose permette la population automatique des informations utilisateur à partir de leur identifiant, cette opération multiplie le nombre de requêtes faites à la base de données, le coût en performances serait alors non négligeable si mis à l'échelle avec des milliers d'utilisateurs simultanés.

Nous avons donc décidé de dupliquer les données d'un utilisateur dans chaque vidéo/commentaire qu'il poste : il s'agit du Subdocument "ReducedUser".

Ainsi, lorsqu'un utilisateur modifie son nom d'utilisateur ou son avatar, cette modification est aussi appliquée sur toutes les vidéos/commentaires, que cet utilisateur a posté. Bien que cette opération puisse sembler lourde aux premiers abords, nous nous sommes mis à la place de l'utilisateur, et nous avons conclu que les modifications de profil sont beaucoup moins fréquentes que la consultation de vidéos, et que cette approche est donc préférable à une jointure dynamique lors du chargement de chaque vidéo/commentaire.

HLS (Http Live Streaming) et options de qualité

Afin de fournir la vidéo à la demande, nous utilisons HLS.

Module Nginx

Nous avons, au début du projet, installé le module de vidéo à la demande lors de la construction de l'image Docker Nginx. Ce module est capable de fournir une vidéo en utilisant la technologie HLS, créant automatiquement le fichier playlist (.m3u8) dès lors que celle-ci a un format compatible.

L'inconvénient de cette méthode est que nous ne pouvons pas lire les fichiers avec un codec autre que H264, H265, et AV1. De plus, la lecture de fichiers en très haute résolution (>2k) peut ne pas fonctionner.

Conversion manuelle

Nous avons donc implémenté une conversion manuelle des vidéos lorsque celles-ci sont mises en ligne en utilisant ffmpeg.

Cette méthode nous oblige à configurer nous-mêmes la génération des différents fichiers .m3u8, de plus, le ré-encodage des vidéos est une charge additionnelle sur les serveurs, malgré cela, les avantages sont nombreux :

- Beaucoup plus de formats vidéo peuvent être mis en ligne car ceux-ci seront tous convertis en un format compatible après la mise en ligne ;
- Les vidéos sont compressées lors de la conversion, ainsi, les données transférées lors de la lecture sont réduites ;
- Les vidéos sont converties en plusieurs résolution :
    - L'utilisateur peut donc choisir la résolution de lecture ;
    - Nous pouvons mieux exploiter les fonctionnalités de HLS en fournissant une résolution "Automatique", qui fera le choix entre les différentes résolutions existantes en fonction du débit de l'utilisateur ;
- Il n'est plus nécessaire d'utiliser le module HLS, nous pouvons directement fournir les fichiers .m3u8 directement par le protocole Http.

Mise en commun des solutions

Afin d'avoir une certaine résilience, nous avons choisi de garder les deux solutions précédemment présentées. Lorsqu'un utilisateur met en ligne une vidéo, la conversion de celle-ci est lancée, mais si jamais cette conversion échoue, l'utilisateur a quand même la possibilité de consulter le fichier vidéo original par l'intermédiaire du module HLS de Nginx (dans la mesure où celle-ci est dans un format compatible).

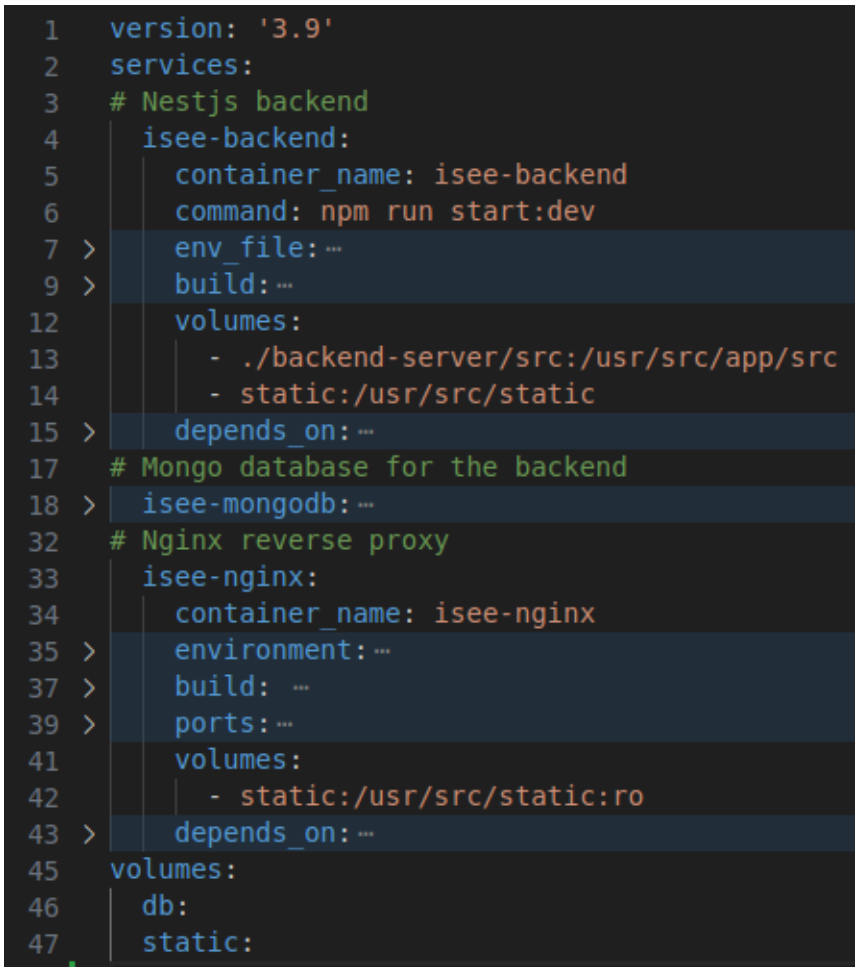
Nous avons donc ajouté le champ "processing" au schéma "Video" afin de garder une trace du processus de ré-encodage, et d'agir en conséquence au niveau du Frontend (par exemple, ne pas proposer les options de qualité sur les vidéos dont le ré-encodage a échoué).

Stockage partagé

Afin de permettre le fonctionnement de notre infrastructure backend, il est nécessaire que notre serveur Backend et notre serveur Nginx aient accès aux fichiers vidéo mis en ligne. Nous avons commencé par un simple transfert d'un serveur à l'autre, mais cette solution n'était pas idéale lorsque le Backend voulait modifier un fichier qu'il avait déjà transmis au serveur Nginx. Il nous a donc fallu mettre un stockage partagé entre nos serveurs.

Avec Docker

Depuis docker, il est facile de mettre en place un stockage partagé, il suffit de monter un même volume nommé/rattaché au stockage local sur chacun des conteneurs.



Le volume “static” est ici déclaré à la ligne 47, et utilisé aux lignes 14 et 42.

Avec Kubernetes

Kubernetes ne permet pas la même flexibilité par design. Un PV (Persistent Volume) rattaché à la machine hôte ne peut être monté que sur un seul Pod à la fois.

Afin de partager un même volume entre différents Pods, nous devons soit utiliser un Provider Cloud en tant que classe de stockage, soit mettre en place un serveur NFS (Network File System). Nous avons donc choisi de déployer internement notre propre serveur NFS, la mise en place de celui-ci ayant rencontré de nombreux problèmes.

Problèmes rencontrés lors de la mise en place du serveur NFS

Lors de la mise en place du serveur, nous ne parvenions pas à lire ou écrire dans des sous dossiers une fois le volume monté sur nos Pods malgré la possibilité de lire/écrire à la racine du volume.

Afin de résoudre ce problème, nous avons dû explorer plusieurs pistes :

Problèmes de connexion

Ce problème pouvait vraisemblablement être écarté, car la connexion au serveur NFS ainsi que le montage du volume fonctionnait.

Problèmes de configuration du conteneur NFS

Nous obtenions les mêmes résultats avec trois images différentes (par erichough, itsthenetwork, et cpuguy83). Nous avons même créé notre propre configuration NFS depuis une image Ubuntu vierge. Le résultat restait le même.

Problème lié à la version NFS

Nous avons ensuite tenté de servir NFS v3 au lieu de NFS v4, sans succès.

Problème lié à Kubernetes

Afin de s'assurer que le driver NFS de Kubernetes n'était pas la cause du problème, nous avons créé un conteneur directement sur Docker, celui-ci rencontrait le même problème.

Suites à cela, nous comprenions que le problème était certainement causé par Docker.

Problème lié au système de fichier

En observant les différents systèmes de fichier présent sur le conteneur docker. Nous avons trouvé la cause potentielle. Nous avons alors essayé d'exporter différents dossiers :

- Un dossier se trouvant à la racine du conteneur, utilisant le système de fichier "overlay"
- Un volume attaché à un dossier "ntfs" de notre hôte
- Un volume attaché à un dossier "ext4" de notre hôte

Ext4 est défini comme étant compatible avec NFS, nous pensions qu'attacher un "bind" volume depuis un dossier du même type fonctionnerait de manière transparente. En recherchant un peu plus, nous nous sommes rendu compte que les volumes "bind" n'utilisaient pas le système de fichier du dossier hôte partagé, mais "virtiofs".

Enfin, nous avons découvert en explorant l'image de "cpuguy83", que créer un volume dans le Dockerfile créait un dossier utilisant ext4. C'est en exportant ce dossier que nous sommes parvenus à faire fonctionner notre serveur NFS.

La solution était donc de créer un volume dans le Dockerfile, puis attacher ce volume à un volume nommé (et non un volume "bind").

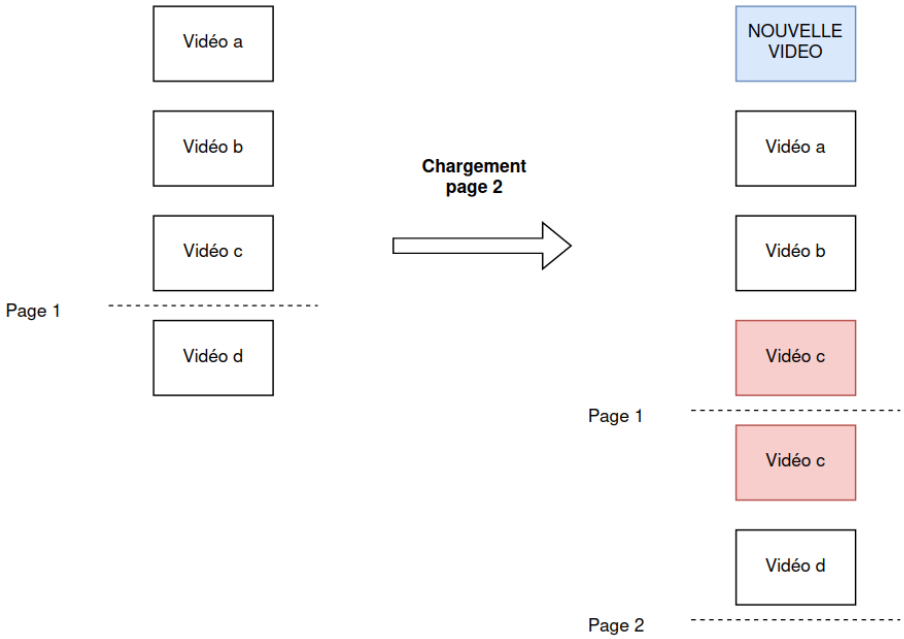
Impact du Frontend sur l'API

La mise en place du Frontend nous a forcé à reconsidérer certaines implémentations.

Pagination intelligente

Lorsque nous avons mis en place la pagination, nous avons utilisé un système de "Requêtes infinies", ce système consiste à charger les nouvelles données lorsque le bas de la page est atteint, et non dans une nouvelle page du front. Cette implémentation apporte un challenge en termes de duplication des données.

Voici le risque encouru si l'on effectue une pagination "naïve" lorsque l'on trie par l'élément le plus récent,



Lors du chargement de la page 1, la vidéo "c" est la troisième vidéo de notre liste. Or, si une nouvelle vidéo est postée entre le chargement de la page 1 et la page 2, la vidéo "c" devient le quatrième élément. Une pagination "naïve" (càd, qui récupère les données "à partir du quatrième élément) causera une duplication de la vidéo c dans notre liste.

Pour éviter ce problème, les requêtes paginées ne récupèrent que les données jusqu'au moment de la "première requête" (le chargement de la page 1). Ainsi, les nouvelles vidéos seront ignorées jusqu'à un rechargement complet de la page.

Hypermédia

La solution présentée précédemment cause cependant un nouveau problème. Les Endpoints paginés prennent maintenant en argument une date de “première requête”. Or, celle-ci peut variée en fonction de la localisation du client, de plus, le front doit gérer la pagination, et garder en mémoire d’autres paramètres tels que la page actuelle et le nombre d’éléments par page. Cela n’est pas dérangeant pour une ressource, mais peut devenir encombrant lorsque les vidéos et les commentaires doivent être paginés.

Afin d’éviter d’avoir à gérer la pagination sur le front, nous nous sommes inspirés d’un concept que nous avons découvert dans une conférence sur les APIs REST, les hypermédias.

Lorsque le frontend veut obtenir une liste de vidéos ou commentaires. Il n’a qu’à appeler l’Endpoint sans aucun paramètre, le backend utilisera alors des valeurs par défaut de page, taille de page, et “date de première requête”. Ces paramètres seront ensuite rajoutés et concaténés en une variable “next”, représentant l’Endpoint à appeler pour obtenir la page suivante.

En faisant cela, la “date de première requête” est basée sur l’horloge du serveur, éliminant alors les soucis de localisation, et facilitant la gestion de la pagination sur le frontend. Cette méthode permet aussi au backend de faire évoluer sa logique interne, sans que le frontend n’ait à changer les paramètres utilisés dans ses requêtes.

Tester les fonctionnalités

Certaines fonctionnalités du frontend (pagination, formatage des nombres, graphes d’évolution dans le dashboard administrateur) peuvent nécessiter un nombre de données assez conséquent, qu’il serait pénible de créer à la main. Nous avons donc ajouté des Endpoints destinés à la création de “fausses” vidéos et “faux” utilisateurs, permettant de choisir le nombre à créer, l’intervalle en heures sur laquelle étendre les dates de création.

Ces données de tests ont été très utiles pour le développement du frontend.

Kubernetes

L'un des derniers challenges de ce projet a été de se familiariser avec Kubernetes, logiciel sur lequel aucun d'entre nous n'avait d'expérience.

Nous avons donc dû nous familiariser avec le vocabulaire de Kubernetes, son fonctionnement, et les différences avec la gestion classique de conteneurs.

La configuration qui en a découlé est la suivante :

- Isee-backend : Deployment
    - Backend-secret : Stockage des différents mots de passe (BDD, mailer) + secret jwt
    - Backend-configmap : Stockage des variables d'environnement
    - Isee-backend-service : service privé
- Isee-nginx : Deployment
    - Isee-nginx-service : service exposé au public
- Isee-mongodb : Stateful Set
    - Mongodb-secret : Stockage des identifiants root
    - Mongodb-service : (Environnement de test uniquement) service exposé au public
    - Mongodb-headless-service : Service privé
    - Isee-mongodb-service

Kubernetes propose nativement des solutions plus adaptées aux déploiements, comme par exemple les secrets, qui sont chiffrés en base64 afin de ne pas stocker des variables d'environnement sensibles en texte brut.

Cela couvre une partie des problèmes rencontrés lors de la mise en place du projet Isee.

Conclusion

Nous touchons à la fin de cette présentation technique du projet Isee. Dans cette partie, nous ferons un rappel de ce qui a été fait, ce qu'on a appris, et les impressions tirées de ce sujet.

Récapitulatif

Afin de mener à bien ce projet, nous avons d'abord mené une face de réflexion et planification approfondie, avant de mettre en place un serveur backend, assurant toute la logique métier et interagissant avec une base de données MongoDB. Nous avons ensuite configuré un serveur Nginx afin de fournir de manière plus optimisée les médias statiques, puis, nous avons créé les fonctionnalités du Frontend avant de styliser ce dernier. Enfin, nous avons créé des configuration Kubernetes pour assurer la haute disponibilité de l'infrastructure, ces configurations étant analogues à notre configuration Docker de développement.

Ce qu'on a appris

Lors de la création de l'infrastructure Isee, nous avons dû nous familiariser avec des technologies parfois seulement connues de nom. Les technologies HLS et Kubernetes ont dû être apprises de zéro pour répondre aux besoins du projet.

Ce projet nous a permis d'améliorer notre capacité à collaborer, nous avons pu apprendre des erreurs commises lors des projets précédents, et avons réduit la taille de notre groupe afin de rendre sa gestion plus facile, nous nous sommes aussi réparti les inconnus de sorte à avoir suffisamment de temps pour les comprendre et les maîtriser, pour ensuite se briefer les uns les autres. Nous nous sommes assurés d'avoir chacun une compréhension correcte du travail de l'autre, et avons communiqué plus souvent sur notre avancée.

Nos impressions

A plusieurs moments lors de ce projet, le sentiment ressenti était un sentiment assez récurrent à ce genre de projets de fin d'années, l'incertitude. Comme ces sujets couvrent souvent plusieurs domaines de l'informatique de par leur vastitude, une partie non-négligeable des tâches à réaliser n'est pas couverte pas les cours suivis durant l'année. Cela met donc à l'épreuve notre curiosité et culture, nous parvenons alors à nous remémorer des langages, outils, protocoles, et services qui pourront selon nous accomplir les tâches demandées, c'est alors à ce moment que nous pouvons ressentir une certaine incertitude, lorsque nous connaissons de nom une solution, mais ne l'avons pas encore mise en pratique pour vérifier si elle répond effectivement à la problématique posée.

Nous estimons ce sentiment d'incertitude parfois désarmant, mais toutefois nécessaire. Plus nous rencontrons ce genre de projets avec un grand nombre d'inconnus, plus nos connaissances s'élargissent. Cela nous permet de pratiquer avec, et apprendre des outils qui, dans le futur, pourront nous permettre de résoudre ces mêmes problématiques si nous venions à les rencontrer de nouveau.

Mot de fin

Nous sommes satisfaits du travail que nous sommes parvenus à fournir dans le cadre de la réalisation de ce projet, et espérons avoir pu vous communiquer notre enthousiasme et passion pour l'ingénierie informatique par l'intermédiaire de ce document.

Cela conclut notre présentation technique du module 3PROJ. Merci de votre attention.