MON BLOG PERSONEL

ANISS MAHFOUDI

Un rapport détaillé sur la conception de mon blog

Octobre 2024

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTINGS

ACRONYMS

DRY Don't Repeat Yourself

API Application Programming Interface

UML Unified Modeling Language

Première partie

PRÉLIMINAIRES

Dans cette première partie, j'établis le cadre nécessaire pour appréhender le projet dans son ensemble. Elle prépare le terrain en fournissant les éléments contextuels et conceptuels indispensables. Vous y trouverez les fondations sur lesquelles repose l'ensemble du rapport, facilitant ainsi la compréhension des sections ultérieures.

1

RÉSUMÉ

1.1 DESCRIPTION DU PROJET

Ce projet s'inscrit dans le cadre de l'obtention de mon titre professionnel et il représente pour moi à la fois une démonstration de mes compétences techniques et une réalisation personnelle. C'est un moyen de partager ma passion et ma curiosité pour le développement et les technologies associées. Ce blog personnel est conçu pour être une plateforme où je peux exprimer mes idées, partager des articles techniques, et explorer des sujets qui me passionnent.
En choisissant de développer un projet full stack, j'ai l'opportunité de mettre en œuvre des technologies modernes et de démontrer une maîtrise technique qui s'étend au-delà du simple développement web. L'intégration de compétences apparentées au DevOps, telles que l'utilisation de Docker pour la containerisation ou la gestion d'une pipeline de deploiement continue, montre ma capacité à gérer l'ensemble du cycle de vie d'une application, de la conception au déploiement.
En fin de compte, ce n'est pas seulement un projet acadé-mique, mais aussi une étape importante dans mon parcours professionnel, me permettant de démontrer ma maîtrise des technologies actuelles et de m'établir comme un développeur confirmé dans le domaine du développement web.

1.2 OBJECTIFS PRINCIPAUX

Les objectifs principaux de ce projet sont :
— Créer une plateforme de blog fonctionnelle et attrayante.
— Démontrer les compétences en développement web à travers des articles et des projets.
— Améliorer la visibilité en ligne et l'employabilité.
— Utiliser des technologies modernes pour le développement et le déploiement.
— Développer des compétences en gestion de projet et en développement logiciel.
— Apprendre et progresser dans le domaine du développement, de la gestion de projet et du DevOps.

1.3 LES COMPÉTENCES MISES EN ŒUVRE

1.3.1 *Compétences Techniques*

- **Développement Backend** : Conception et développement de l'architecture backend, y compris la gestion de contenu et la configuration des bases de données.
- **Développement Frontend** : Création d'interfaces utilisateur dynamiques et réactives, avec une attention particulière à l'optimisation pour le référencement naturel (SEO).
- **Intégration et Déploiement** : Mise en place de processus d'intégration continue et de déploiement continu (CI/CD) pour assurer une livraison rapide et fiable des fonctionnalités.
- **Sécurité Informatique** : Implémentation de mesures de sécurité pour protéger les données et les communications, y compris la gestion des accès et la protection contre les menaces.
- **Choix des Technologies** : Sélection de technologies adaptées aux besoins du projet pour garantir efficacité, évolutivité et maintenabilité.

1.3.2 *Compétences en Gestion de Projet*

- **Gestion Agile** : Application de méthodologies agiles pour planifier, exécuter et suivre le projet, en favorisant l'adaptabilité et la collaboration.
- **Planification et Organisation** : Capacité à structurer le projet en phases et à gérer les ressources et les délais de manière efficace.
- **Gestion des Risques** : Identification et gestion proactive des risques pour minimiser les impacts négatifs sur le projet.

1.3.3 *Compétences en Développement Personnel*

- **Apprentissage Autonome** : Capacité à acquérir de nouvelles connaissances et compétences de manière autonome pour s'adapter aux évolutions technologiques.
- **Résolution de Problèmes** : Aptitude à identifier, analyser et résoudre les problèmes techniques rencontrés au cours du projet.
- **Adaptabilité** : Capacité à s'adapter rapidement aux changements et à intégrer de nouvelles idées et technologies.

1.3.4 *Compétences en Conception et Modélisation*

- **Analyse des Besoins** : Identification et analyse des besoins fonctionnels et non fonctionnels pour définir les spécifications du projet.

— **Modélisation de Données** : Conception de modèles de données pour structurer et organiser les informations de manière efficace.
— **Conception d'Architecture** : Élaboration d'architectures système robustes et évolutives pour répondre aux exigences du projet.

1.3.5 *Compétences en Tests et Validation*

— **Tests Unitaires et d'Intégration** : Conception et exécution de tests pour vérifier la qualité et la fiabilité du code, assurant ainsi le bon fonctionnement des fonctionnalités développées.
— **Tests de Performance** : Évaluation des performances du système pour garantir des temps de réponse optimaux et une utilisation efficace des ressources.
— **Tests de Charge** : Évaluation de la résistance du système à des charges importantes pour s'assurer de sa stabilité et de ses performances.

1.4 STRUCTURE DU RAPPORT

Ce rapport est structuré en plusieurs parties distinctes, chacune visant à fournir une compréhension approfondie des différents aspects du projet. Voici un aperçu de la structure du rapport et des parties à venir :

1.4.1 *Partie 1 : Préliminaires*

Cette partie introductive établit le cadre du projet, présentant le contexte, les objectifs, et les compétences mises en œuvre. Elle fournit une vue d'ensemble essentielle pour comprendre la portée et l'importance du projet.

1.4.2 *Partie 2 : Conception et Méthodologies*

Je détaille ici la conception du projet, y compris l'analyse des besoins, la définition du produit minimum viable (MVP), et l'architecture du système. Je présente également les méthodologies de développement que j'ai utilisées pour structurer et gérer le projet.

1.4.3 *Partie 3 : Développement*

Cette partie couvre le processus de développement technique du projet. Elle décrit en détail la mise en place de l'environnement de développement, le développement du backend avec Strapi, l'implémen-

tation du frontend avec Next.js, et l'intégration des fonctionnalités clés du blog.

1.4.4 *Partie 4 : Déploiement*

Je présente ici le processus de déploiement dans son ensemble. Cela inclut la configuration de l'environnement de production, le déploiement des conteneurs Docker, la mise en place de Nginx comme reverse proxy, et l'implémentation de la sécurité avec SSL via Certbot. Cette partie aborde également les stratégies de monitoring, les tests en production, et la mise en place d'un pipeline de déploiement continu.

1.4.5 *Partie 5 : Etude de cas et Conclusion*

Cette partie finale du rapport se concentre sur une étude de cas pratique et la conclusion générale du projet. L'étude de cas illustre l'application concrète des solutions développées, en mettant en avant les résultats obtenus et les leçons apprises. La conclusion récapitule les principaux points abordés dans le rapport, évalue l'atteinte des objectifs initiaux et propose des perspectives pour des travaux futurs. Cette section vise à démontrer l'impact et la pertinence du projet dans un contexte réel, tout en offrant une réflexion critique sur le travail accompli.

Deuxième partie

CONCEPTION ET MÉTHODOLOGIES

Dans cette partie, je vous invite à découvrir comment j'ai abordé la conception et l'architecture de mon blog, ainsi que la méthodologie de travail qui a guidé chaque étape du projet. Ces étapes sont cruciales et déterminent la qualité du produit final ainsi que tout le processus de développement.

2

CONCEPTION DU PROJET

2.1 BESOINS FONCTIONNELS

2.1.1 *Objectifs Principaux*

Les objectifs principaux, ont déjà été exprimés dans le chapitre 1.

— **Partage de Contenu** : Fournir une plateforme pour partager des articles sur des sujets qui me passionnent, permettant d'engager et d'informer un public intéressé.
— **Visibilité Numérique** : Établir une présence numérique pour accroître ma visibilité en ligne, en utilisant le blog comme un outil de communication et de marketing personnel.
— **Démonstration de Compétences** : Utiliser le blog pour démontrer mes compétences en rédaction, en gestion de contenu, et en développement web, renforçant ainsi ma crédibilité professionnelle.
— **Apprendre et évoluer** : Utiliser le blog comme un laboratoire pour tester de nouvelles idées et apprendre de nouvelles technologies.

2.1.2 *Fonctionnalités et Interactions Utilisateur*

Les fonctionnalités et interactions utilisateur sont conçues pour offrir une expérience enrichissante et personnalisée aux visiteurs du blog.

— **Publication d'Articles** : En tant que propriétaire du blog, je peux publier des articles, partageant ainsi mes idées et mes connaissances avec les lecteurs.
— **Consultation de Contenu** : Les lecteurs peuvent facilement accéder aux articles publiés, explorant divers sujets d'intérêt.
— **Recherche d'Articles** : Les utilisateurs peuvent rechercher des articles par tag et par nom, facilitant l'accès à des contenus spécifiques.
— **Personnalisation de l'Interface** : Les utilisateurs ont la possibilité de choisir entre un thème sombre et un thème clair, personnalisant ainsi leur expérience de lecture.
— **Commentaires** : Dans des versions ultérieures, les lecteurs pourront commenter les articles et se connecter pour une expérience plus interactive.

— **Flux RSS** : Un flux RSS sera mis en place pour permettre aux utilisateurs de suivre les mises à jour du blog de manière pratique.

Un flux RSS (Really Simple Syndication) est un format de fichier XML utilisé pour partager des mises à jour de contenu web.

2.1.3 *Gestion de Contenu*

La gestion de contenu est un aspect essentiel du projet, permettant une organisation efficace et flexible des articles et des médias associés.

— **Utilisation d'un CMS Open Source et Headless**[1] : Le contenu est géré via un CMS open source, garantissant la propriété des données et offrant une interface conviviale pour la gestion des articles et des médias.

— **Articles en Markdown** : Les articles seront écrits en Markdown, permettant une gestion simplifiée du contenu et une mise en page flexible.

Le Markdown est un langage de balisage léger qui permet de formater du texte en utilisant une syntaxe simple et lisible.

— **Inclusion de Médias** : Les articles peuvent inclure des images pour enrichir le contenu visuel et illustrer les points clés. Les images peuvent être téléchargées et intégrées directement dans le texte.

— **Fonctionnalités de Brouillon et de Publication** : Les articles peuvent être enregistrés en tant que brouillons, permettant des révisions et des modifications avant la publication. Une fois finalisés, les articles peuvent être publiés pour être consultés par les lecteurs.

— **Gestion des Auteurs** : Actuellement, je suis le seul auteur, ce qui simplifie la gestion des droits et des permissions. Cependant, le système est conçu pour évoluer et accueillir potentiellement d'autres contributeurs à l'avenir.

— **Archivage des Articles** : Les articles peuvent être archivés en les repassant en mode brouillon, permettant de les retirer temporairement de la publication tout en conservant leur contenu pour une utilisation future.

2.2 BESOINS NON FONCTIONNELS

Les besoins non fonctionnels du projet visent à garantir une performance optimale, une sécurité adéquate, et une expérience utilisateur de haute qualité.

— **Performance** : Assurer un temps de chargement le plus court possible pour améliorer le référencement et offrir une expérience

1. Un Headless CMS (Content Management System) est un système de gestion de contenu qui déconnecte le backend (où le contenu est créé et stocké) du frontend (où le contenu est présenté aux utilisateurs). Contrairement aux CMS traditionnels, qui gèrent à la fois le contenu et sa présentation, un headless CMS se concentre uniquement sur la gestion du contenu et expose une API (souvent RESTful ou GraphQL) pour permettre à divers clients (sites web, applications mobiles, etc.) de récupérer et d'afficher ce contenu.

utilisateur fluide. Cela inclut l'optimisation des ressources et l'utilisation de techniques de mise en cache.
— **Sécurité** : Bien que les données ne contiennent pas d'informations privées, l'environnement de production doit être sécurisé, notamment en raison de son déploiement sur un VPS. Cela inclut la mise en place de pare-feu et de protocoles de sécurité pour protéger l'infrastructure.
— **Scalabilité** : Le système doit être conçu pour s'adapter facilement à une augmentation du nombre d'utilisateurs ou de contenu, garantissant ainsi la pérennité du projet à long terme.
— **Disponibilité** : Le service doit être disponible en permanence, avec des interruptions minimales pour les mises à jour ou la maintenance, assurant ainsi une fiabilité constante pour les utilisateurs.
— **Compatibilité** : Le site doit être full responsive, compatible avec une large gamme d'appareils, y compris les ordinateurs, tablettes et téléphones, pour atteindre le plus grand nombre d'utilisateurs possible.
— **Optimisation pour le Référencement** : Le projet doit être optimisé pour le référencement, en utilisant des pratiques SEO modernes pour améliorer la visibilité et l'accessibilité du contenu.
— **Accessibilité** : Conformité aux normes d'accessibilité pour garantir que le site est utilisable par tous, y compris les personnes handicapées, en respectant les directives d'accessibilité du contenu web (WCAG)[2].
— **Maintenance et Support** : Planification pour la maintenance régulière et le support technique, assurant que le système reste à jour et fonctionnel.
— **Localisation** : Support multilingue pour atteindre un public plus large, permettant aux utilisateurs de naviguer et d'interagir avec le contenu dans leur langue préférée.
— **Conformité Légale** : Respect des réglementations en matière de protection des données, comme le RGPD, pour garantir la confidentialité et la sécurité des informations des utilisateurs.

Pour ce projet, aucune donnée personnelle n'est collectée ni traitée, donc aucun registre des données n'est à tenir.

2.2.1 *Diagrammes de Cas d'Utilisation*

Le diagramme de cas d'utilisation permettra de visualiser les principales fonctionnalités du projet. Les sous fonctionnalités (ajouter une image, publier un brouillon, etc.) ne sont pas représentées dans le diagramme dans la mesure où elles sont considérées comme des détails d'implémentation et font partie intégrante du CMS et de la gestion du contenu.

2. https://www.w3.org/WAI/standards-guidelines/wcag/

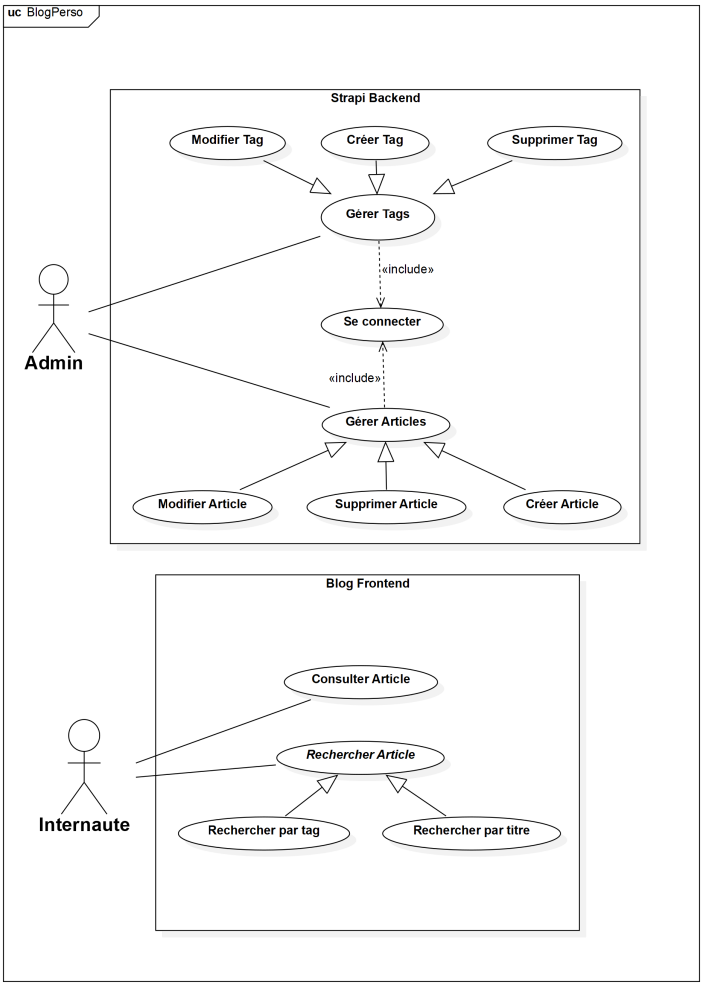


FIGURE 1 – Diagramme de Cas d'Utilisation

2.3 MINIMUM VIABLE PRODUCT (MVP)

Le Minimum Viable Product (MVP) du projet vise à établir une première version fonctionnelle du blog, permettant de publier et de consulter des articles simples rédigés en Markdown. L'objectif principal est de créer une plateforme accessible à tous sur le web, tout en offrant une interface de gestion de contenu efficace pour l'administrateur.

2.3.1 *Objectif Principal*

L'objectif du MVP est de fournir une version initiale du blog qui permet la présentation d'articles simples, accessibles à tous les utilisateurs sur le web. Cette version doit démontrer la capacité à gérer et afficher du contenu de manière efficace.

2.3.2 *Fonctionnalités Essentielles*

Les fonctionnalités minimales nécessaires pour le MVP incluent :
— **Gestion de Contenu** : Possibilité pour l'administrateur de se connecter au backend et de gérer le contenu depuis n'importe où. Les modifications de contenu doivent apparaître instantanément sur le site.
— **Affichage des Articles et Tags** : Les articles doivent être affichés correctement sur le site, avec une gestion efficace des tags associés.
— **Gestion des Articles et Tags** : Capacité à créer, modifier et supprimer des articles et des tags via le backend.

2.3.3 *Public Cible*

Le MVP cible deux groupes d'utilisateurs principaux :
— **Lecteurs** : Utilisateurs qui consultent les articles publiés sur le blog.
— **Administrateur** : L'administrateur (moi) qui gère le contenu du site, notamment les articles et les tags.

2.3.4 *Technologies Utilisées*

Le développement du MVP repose sur les technologies suivantes :
— **VPS** : Utilisation d'un serveur privé virtuel pour héberger l'application.
— **Docker** : Environnement de containerisation pour déployer les services de manière isolée et efficace.
— **Next.js** : Framework utilisé pour le développement du frontend, offrant des fonctionnalités avancées de rendu côté serveur.
— **Strapi** : CMS headless utilisé pour le backend, permettant une gestion flexible du contenu.

2.3.5 *Critères de Succès*

Les indicateurs de succès pour le MVP incluent :
— **Accessibilité** : Le site doit être accessible via un nom de domaine, avec une connexion sécurisée (HTTPS).
— **Consultabilité** : Le site (Next.js) doit être consultable partout, affichant correctement les articles et les tags.
— **Accessibilité du Backend** : Le panneau d'administration du backend (Strapi) doit être accessible par l'administrateur depuis n'importe où, via un sous-domaine sécurisé.

2.4 ARCHITECTURE DU SYSTÈME

L'architecture du projet étant relativement simple, la création d'un diagramme d'architecture n'était pas nécessaire. Le travail s'est concentré sur le déploiement physique des composants, représenté par un diagramme de déploiement qui fournit déjà une vue complète des interactions et de l'emplacement des éléments. Par conséquent, le diagramme de déploiement est suffisant pour comprendre l'organisation du système, et un diagramme d'architecture supplémentaire n'aurait pas apporté de valeur ajoutée significative.

2.4.1 *Diagramme de déploiement*

Ce diagramme représente l'architecture initiale du projet. La version finale de l'infrastructure inclura également une solution de monitoring, qui sera détaillée ultérieurement dans ce rapport.

Voici donc le diagramme de déploiement du système :

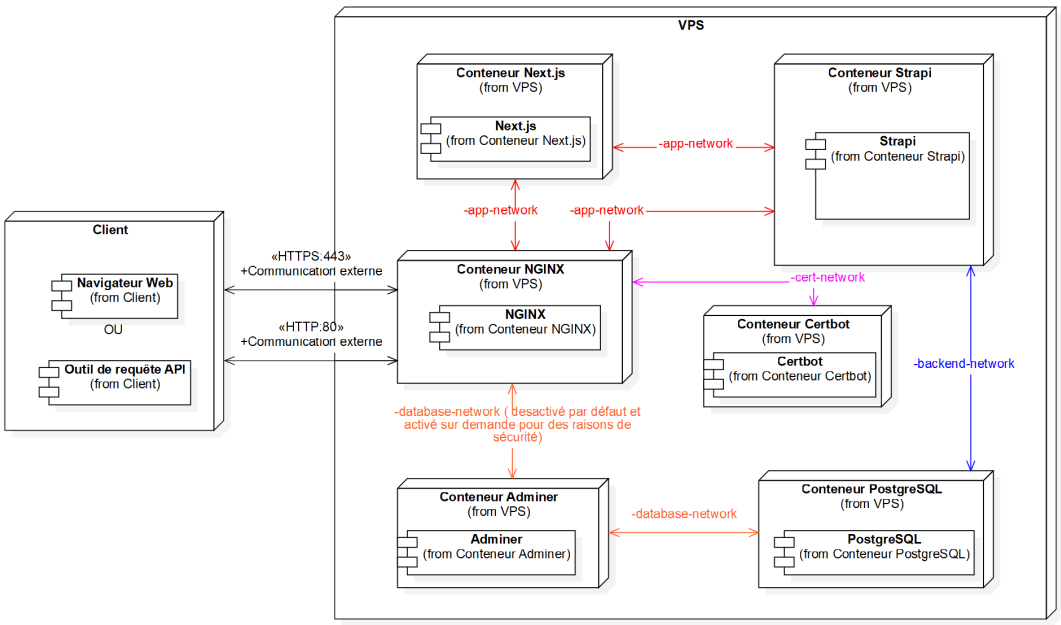


FIGURE 2 – Diagramme de déploiement du système

Description du diagramme de déploiement :
Dans la mesure ou le diagramme qui est fourni pourrait etre peu lisible, voici une description textuelle des composants et des relations :

1. **Client :** Le client est une entité externe qui peut communiquer avec le serveur via deux méthodes :
   — *Navigateur Web :* Le navigateur web envoie des requêtes vers le serveur via les protocoles HTTP (port 80) ou HTTPS (port 443).
   — *Outil de requête API :* Il permet d'envoyer des requêtes directement à l'API via les mêmes protocoles (HTTP et HTTPS).
2. **VPS :** Le serveur virtuel est l'hôte des conteneurs Docker et des réseaux qui permettent l'interaction entre les composants.
3. **Conteneurs sur le VPS :**

— *Conteneur NGINX :* Ce conteneur agit comme reverse proxy[3] et gère le routage des requêtes provenant du client vers les services internes. Il utilise les protocoles HTTP (port 80) et HTTPS (port 443) pour la communication externe. Les requêtes HTTP sont redirigées vers HTTPS.
— *Conteneur Next.js :* Ce conteneur héberge l'application frontend. Il est relié au `app-network` pour permettre la communication avec NGINX et le backend Strapi.
— *Conteneur Strapi :* Héberge l'API Strapi (backend), accessible via `app-network` pour communiquer avec Next.js et Nginx (le panel admin est accessible depuis l'extérieur).
— *Conteneur Certbot :* Il permet la mise en place de certificats SSL sur l'application et leur renouvellement automatique, il ne sert que ponctuellement.
— *Conteneur PostgreSQL :* Ce conteneur contient la base de données PostgreSQL, utilisée par Strapi. Il est relié au `database-network` afin de pouvoir être administré via Adminer, ainsi qu'au `backend-network` pour la communication avec Strapi.
— *Conteneur Adminer :* Adminer est un outil d'administration de base de données qui se connecte à PostgreSQL via `database-network`. Cependant, le réseau et le conteneur sont désactivés par défaut et activés sur demande pour des raisons de sécurité afin de réduire la surface d'attaque.

Les certificats SSL permettent la mise en place de communications chiffrées par le biais du protocole HTTPS.

La surface d'attaque représente l'ensemble des points d'entrée potentiels qu'un attaquant pourrait exploiter dans un système. Réduire cette surface limite les opportunités pour les cyberattaques.

4. **Réseaux utilisés :**
   — *app-network :* Utilisé pour la communication entre NGINX, Next.js, et Strapi.
   — *backend-network :* Utilisé pour la communication entre Strapi et la base de données PostgreSQL.
   — *database-network :* Utilisé pour la connexion entre Adminer et PostgreSQL (désactivé par défaut).
   — *cert-network :* Utilisé pour la communication entre NGINX et Certbot pour gérer les certificats SSL.
5. **Flux de communication externe :** Le navigateur web ou l'outil de requête API dans le client communique avec NGINX via HTTP (port 80) ou HTTPS (port 443). NGINX fait ensuite suivre les requêtes vers les autres conteneurs - en forçant le protocole HTTPS - via les réseaux internes selon le nom de domaine demandé par le client.

L'architecture décrite dans ce diagramme de déploiement met en évidence une isolation des services, tant d'un point de vue logique que réseau. Chaque conteneur est responsable d'une fonctionnalité spécifique (frontend, backend, base de données, proxy, gestion des certificats, etc.) et est connecté à d'autres services uniquement via des réseaux Docker bien définis. Cette segmentation garantit que les services ne peuvent interagir que de manière contrôlée à travers les réseaux appropriés (par exemple, `app-network` pour l'interaction entre NGINX, Next.js, et Strapi, ou `database-network` pour les connexions avec la base de données).

Bénéfices de cette architecture :

— **Sécurité accrue** : L'isolation des services empêche un accès direct entre les conteneurs qui ne doivent pas communiquer, limitant ainsi

3. Un reverse proxy est un serveur intermédiaire qui reçoit les requêtes des clients et les transmet aux serveurs appropriés, agissant comme un point d'entrée unique pour plusieurs services.

les vecteurs potentiels d'attaques. Le seul point d'entrée du systeme est le domaine accessible depuis l'extérieur, toutes les autres interactions sont internes, et NGINX à lui seul gère le routage des requêtes vers les services internes.
— **Facilité de gestion** : En segmentant chaque service, il devient plus simple d'administrer et de mettre à jour les composants sans affecter les autres parties du système. Cela favorise un déploiement continu et des mises à jour sans interruption.
— **Scalabilité** : L'isolation permet de facilement faire évoluer chaque service de manière indépendante, en fonction des besoins de charge. Par exemple, si l'application Next.js a besoin d'être mise à l'échelle, cela peut être fait sans affecter la base de données ou l'API Strapi.
— **Résilience** : La séparation des services garantit qu'en cas de panne d'un conteneur, les autres continuent de fonctionner normalement, améliorant ainsi la robustesse de l'infrastructure globale.
— **Conformité** : L'isolation des services permet de facilement mettre en place des mesures de sécurité supplémentaires, comme des pare-feux, en configurant les réseaux Docker de manière appropriée.

Ainsi, cette approche modulaire et isolée optimise à la fois la sécurité, la gestion et la performance du système dans son ensemble et répond à la plupart des besoins non fonctionnels énoncés précédemment.

2.5 MODÉLISATION DE LA BASE DE DONNÉES

2.5.1 *Modèle Entité-Association*

Dans le cadre de ce projet, j'ai choisi de me concentrer uniquement sur la création d'un modèle entité-association pour la modélisation de la base de données. Cette décision est motivée par l'utilisation de Strapi en tant que système de gestion de contenu.

Strapi offre une gestion intégrée et automatisée de l'implémentation des modèles de données dans la base de données. Grâce à ses fonctionnalités avancées, Strapi prend en charge la création et la gestion des tables et des relations dans le système de gestion de base de données sous-jacent, éliminant ainsi le besoin de concevoir un modèle logique de données (MLD) séparé.

Les avantages de cette approche incluent :
— **Simplicité** : En s'appuyant sur Strapi, le processus de modélisation est simplifié, ce qui permet de se concentrer sur la définition des entités et de leurs relations sans se soucier des détails d'implémentation.
— **Efficacité** : Strapi automatise la génération des schémas de base de données, réduisant ainsi le temps et les efforts nécessaires pour configurer et maintenir la base de données.
— **Flexibilité** : Les modifications apportées au modèle entité-association peuvent être facilement intégrées et mises à jour dans la base de données via Strapi, assurant une adaptabilité continue aux besoins du projet.

— **Cohérence** : En centralisant la gestion des données dans Strapi, on garantit une cohérence entre le modèle conceptuel et l'implémentation physique.

En conclusion, l'utilisation de Strapi comme CMS permet de rationaliser le processus de modélisation de la base de données, tout en assurant une gestion efficace et cohérente des données.

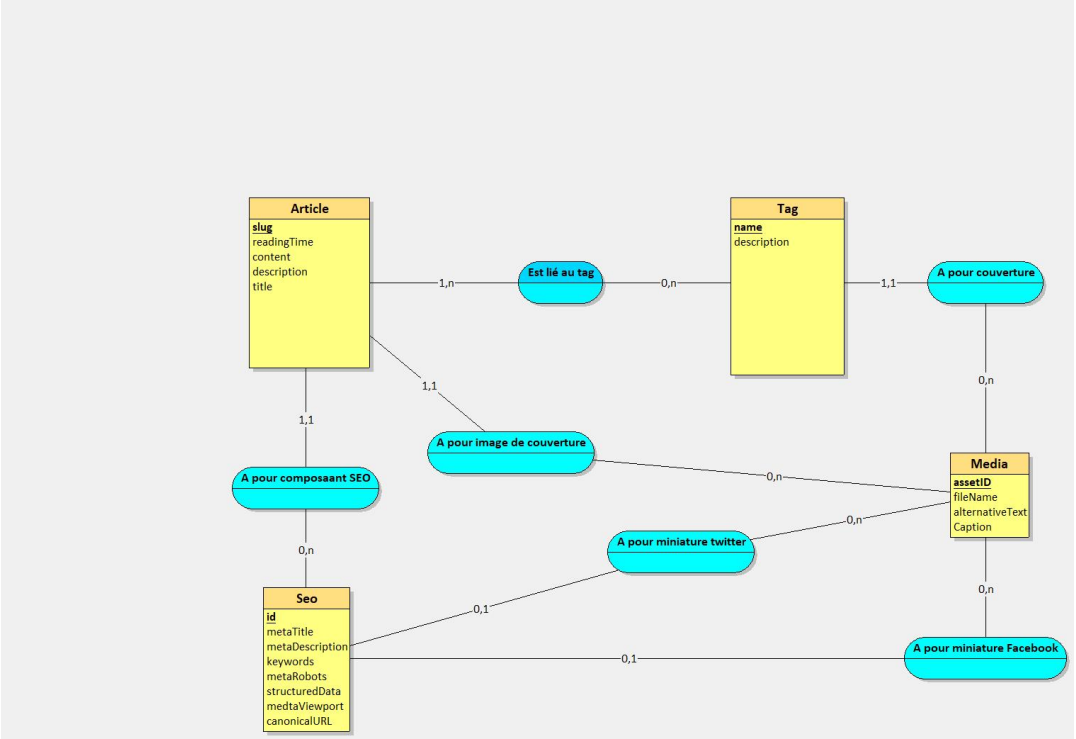


FIGURE 3 – Modèle Entité-Association du projet

2.5.2 *Description du Modèle Entité-Association*

Voici une description détaillée des entités et de leurs relations :

2.5.2.1 *Entités*

article

— Attributs : slug, readingTime, content, description, title
— Relations :
  — Est lié à un ou plusieurs **Tag**
  — A un composant **SEO**
  — A une image de couverture **Media**
  — A une miniature Twitter **Media**
  — A une miniature Facebook **Media**

tag

— Attributs : name, description

- Relations :
  - Est lié à 0 ou plusieurs **Article**
  - A une couverture **Media**

media
- Attributs : assetID, fileName, alternativeText, Caption
- Relations :
  - Peut être une couverture pour un **Tag**
  - Peut être une image de couverture, une miniature Twitter ou Facebook pour un **Article**
  - Peut être une image de couverture pour un **Tag**

seo
- Attributs : id, metaTitle, metaDescription, keywords, metaRobots, structuredData, mediaViewport, canonicalURL
- Relations :
  - Est un composant SEO pour un **Article**

Les types uniques ne sont pas explicitement représentés dans ce modèle E-A car ils sont gérés par Strapi et se représentent difficilement dans un modèle E-A. Il est cependant important de les documenter et c'est ce que je vais faire dans la section suivante.

2.5.2.2 *Types Uniques*

Il est prevus dans un premiers temps, deux types unique pour générer le contenu de certaines pages de manière dynamique :
- **About** (Pour la page "A Propos")
  - Contient un champ `content` qui permet de renseigner le contenu de la page
  - A un composant **SEO**
- **HomeBlogPage** (Pour la page d'accueil du blog)
  - Contient un `featured article` qui est un article sélectionné pour être mis en avant sur la page d'accueil
  - A un composant **SEO**

2.6 CONCEPTION VISUELLE ET ARCHITECTURE DU SITE

2.6.1 *Maquette du Site*

La maquette a été réalisée à l'aide de Figma. Les thèmes clair et sombre sont prévus, ainsi que le mode mobile. Voici un aperçu de la maquette :

Les planches détaillées sont disponibles dans l'annexe A de ce rapport.

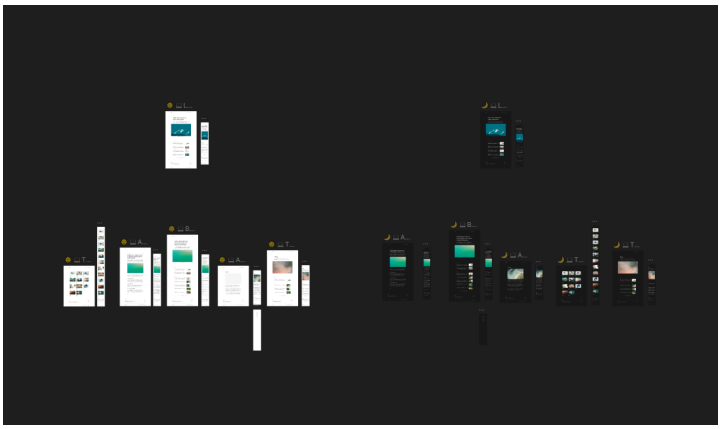


FIGURE 4 – Maquette du Site

2.6.2 *Arborescence du Site*

L'arborescence du site est une représentation graphique de la structure du site web, montrant les relations entre les différentes pages et leur organisation hiérarchique.

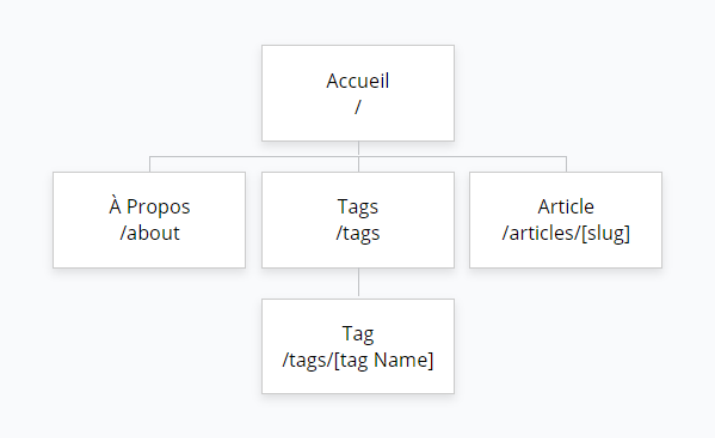


FIGURE 5 – Arborescence du Site

La sitemap du site est structurée de manière hiérarchique comme suit :

- **Accueil (/)**
  - La page d'accueil du site, point d'entrée principal pour les utilisateurs.
- **À Propos (/about)**
  - Une page dédiée à ma présentation et celle du blog.
- **Tags (/tags)**
  - Une page listant tous les tags disponibles, permettant aux utilisateurs de naviguer par catégories de contenu.
- **Tag (/tags/[tagName])**
  - Une page dynamique pour chaque tag spécifique, affichant les articles associés à ce tag.

- **Article (/articles/[slug])**
  - Une page dynamique pour chaque article, identifiée par un slug unique, permettant d'accéder au contenu détaillé de chaque publication.

3

MÉTHODOLOGIE DE DÉVELOPPEMENT

3.1 APPROCHE FEATURE DRIVEN DEVELOPMENT (FDD)

Pour le développement, j'ai adopté une approche s'inspirant du Feature Driven Development (FDD). Cette méthodologie a été choisie pour sa capacité à se concentrer sur les fonctionnalités individuelles, ce qui est particulièrement adapté à un projet personnel en constante évolution.

3.2 OUTILS DE GESTION DE PROJET

Pour faciliter la gestion du projet selon les principes du FDD, j'ai utilisé les outils suivants :

— **Trello** : Pour la gestion de la liste des fonctionnalités et le suivi de leur progression.
— **GitHub** : Pour le contrôle de version, avec des branches spécifiques pour chaque fonctionnalité.
— **StarUML** : Pour la modélisation globale et la conception des fonctionnalités.
— **Anytype** : Pour la gestion de la documentation et la gestion de projet.

3.3 PHASES DE DÉVELOPPEMENT

Le développement du projet a suivi les phases caractéristiques du FDD :

3.3.1 *Phase 1 : Développement d'un Modèle Global*

— Création d'un modèle de domaine pour le blog
— Identification des principales entités et leurs relations

3.3.2 *Phase 2 : Construction de la Liste des Fonctionnalités*

— Identification et description détaillée de chaque fonctionnalité
— Priorisation des fonctionnalités selon leur importance pour le MVP

3.3.3 *Phase 3 : Planification par Fonctionnalité*

— Estimation du temps de développement pour chaque fonctionnalité
— Création d'un calendrier de développement

3.3.4 *Phase 4 : Conception et Construction par Fonctionnalité*

— Conception détaillée de chaque fonctionnalité
— Développement, test et intégration de chaque fonctionnalité

3.4 PRATIQUES DE DÉVELOPPEMENT

En accord avec les principes du FDD, j'ai adopté les pratiques suivantes :
— **Inspections de Code** : Révision régulière du code pour assurer sa qualité et sa cohérence.
— **Builds Réguliers** : Intégration continue des fonctionnalités développées pour maintenir une version stable du projet.
— **Gestion de Configuration** : Utilisation de Git pour suivre les changements et gérer les versions du code.

3.5 GESTION DES VERSIONS

La gestion des versions a été adaptée aux principes du FDD :
— **Main** : Branche principale contenant le code stable et testé.
— **Feature Branches** : Une branche par fonctionnalité en cours de développement.

3.6 CONCLUSION

L'adoption d'une approche inspirée du Feature Driven Development a permis de développer mon blog de manière structurée et orientée vers les fonctionnalités. Les pratiques et outils mis en place ont contribué à maintenir un code de qualité et à gérer efficacement le cycle de vie du développement de chaque fonctionnalité. Le fait que je sois à la fois le client et le réalisateur du projet, combiné au nombre limité de fonctionnalités à implémenter, se prêtait particulièrement bien à cette approche, permettant une flexibilité et une adaptation rapide tout au long du processus de développement.

Troisième partie

DÉVELOPPEMENT

Cette partie détaille le processus de développement du blog, en mettant l'accent sur les choix techniques, les défis rencontrés et les solutions mises en œuvre. Elle offre un aperçu approfondi des étapes de réalisation concrète du projet, depuis la mise en place de l'environnement de développement jusqu'à l'implémentation des fonctionnalités clés.

4 ENVIRONNEMENT DE DÉVELOPPEMENT

4.1 INTRODUCTION

Dans ce chapitre, je décris l'environnement de développement utilisé pour le projet.

4.2 CONFIGURATION MATÉRIELLE

Je dispose de deux ordinateurs, un fixe et un portable. Mon ordinateur fixe est un ordinateur très performant (Processeur Ryzen 7 5800X, 32GB de RAM) et mon ordinateur portable l'est beaucoup moins (uniquement 8GB de RAM et un processeur d'ancienne génération à deux cœurs). Pour ces raisons j'ai principalement développé sur mon ordinateur fixe.

4.3 CONFIGURATION LOGICIELLE

Le système d'exploitation principal est Windows, couplé à l'utilisation de Windows Subsystem for Linux (WSL)[1] avec une distribution Ubuntu. Cette configuration permet de bénéficier des avantages des deux environnements, Windows pour l'interface utilisateur et Ubuntu pour les outils de développement Linux.

1. https://learn.microsoft.com/fr-fr/windows/wsl/

4.4 OUTILS ET TECHNOLOGIES

4.4.1 *Visual Studio Code*

Visual Studio Code (VSCode) est l'environnement de développement intégré (IDE) choisi pour ce projet. Avec son large écosystème d'extensions, il permet une grande personnalisation et une intégration avec différents outils et langages de programmation.

4.4.2 *Node.js*

Node.js est un environnement d'exécution JavaScript côté serveur, basé sur le moteur V8 de Chrome. Node.js a été choisi pour principalement sa grande communauté, ce qui en fait un runtime stable et documenté.

4.4.3 *TypeScript*

TypeScript est un superset[2] typé de JavaScript qui se compile en JavaScript pur. Il ajoute des types statiques optionnels, des classes, et d'autres fonctionnalités à JavaScript. TypeScript a été utilisé pour améliorer la maintenabilité du code, réduire les erreurs potentielles, et faciliter le développement d'applications à grande échelle.

Bien que l'ecriture du code soit plus lourde, les avantages en termes de maintenabilité du code et de reduction des erreurs sont plus que nécessaires.

4.4.4 *Yarn*

Yarn est un gestionnaire de paquets pour JavaScript, développé par Facebook en collaboration avec d'autres entreprises. Il constitue une alternative à npm, le gestionnaire de paquets officiel de Node.js. Yarn a été choisi pour sa vitesse et sa gestion efficace des dépendances.

Strapi recommande fortement l'utilisation de Yarn pour la gestion des dépendances de ses projets.

2. Un superset est un langage de programmation qui étend un autre langage en ajoutant de nouvelles fonctionnalités tout en conservant la compatibilité avec le langage d'origine.

5 BACKEND AVEC STRAPI

5.1 INTRODUCTION À STRAPI

Strapi est un système de gestion de contenu (CMS) headless open-source basé sur Node.js. Il offre une grande flexibilité et permet de créer rapidement des API personnalisées.

5.2 INSTALLATION ET CONFIGURATION

Strapi offre deux types d'environnements distincts pour le développement et la production :

— **Environnement de développement** : Utilisé lors de la phase de création et de test de l'application. Il est optimisé pour le débogage et offre des fonctionnalités comme le hot-reloading[1] pour faciliter le développement. En développement, il est possible d'ajouter des types de contenu et de les éditer.

L'environnement de production ne sera utilisé que lors du déploiement du projet sur le serveur.

— **Environnement de production** : Conçu pour le déploiement final de l'application. Il est optimisé pour les performances et la sécurité, avec des fonctionnalités telles que la mise en cache et la compression des assets. En production, les types de contenu sont figés et ne peuvent pas être modifiés, seules les données peuvent être modifiées.

Dans cette partie nous travaillerons bien entendu dans l'environnement de développement.

5.2.1 *Installation de Strapi*

Pour créer un projet Strapi, nous utiliserons le CLI de Strapi. Voici ce que cela donne :

CLI signifie Command Line Interface, ou Interface en Ligne de Commande. C'est un outil qui permet d'interagir avec un logiciel via des commandes textuelles plutôt que par une interface graphique.

Voila un résumé des différentes commandes saisies et des options choisies :

1. `nvm use 20`
   Utilise la version 20 de Node.js via Node Version Manager (nvm).
2. `yarn create strapi-app`
   Utilise Yarn pour créer une nouvelle application Strapi.
3. **Nom du projet** : `backend`
   On choisit le nom du projet, ici `backend`.

1. Le hot-reloading est une technique qui permet de recharger automatiquement les modifications du code sans redémarrer l'application, accélérant ainsi le processus de développement.

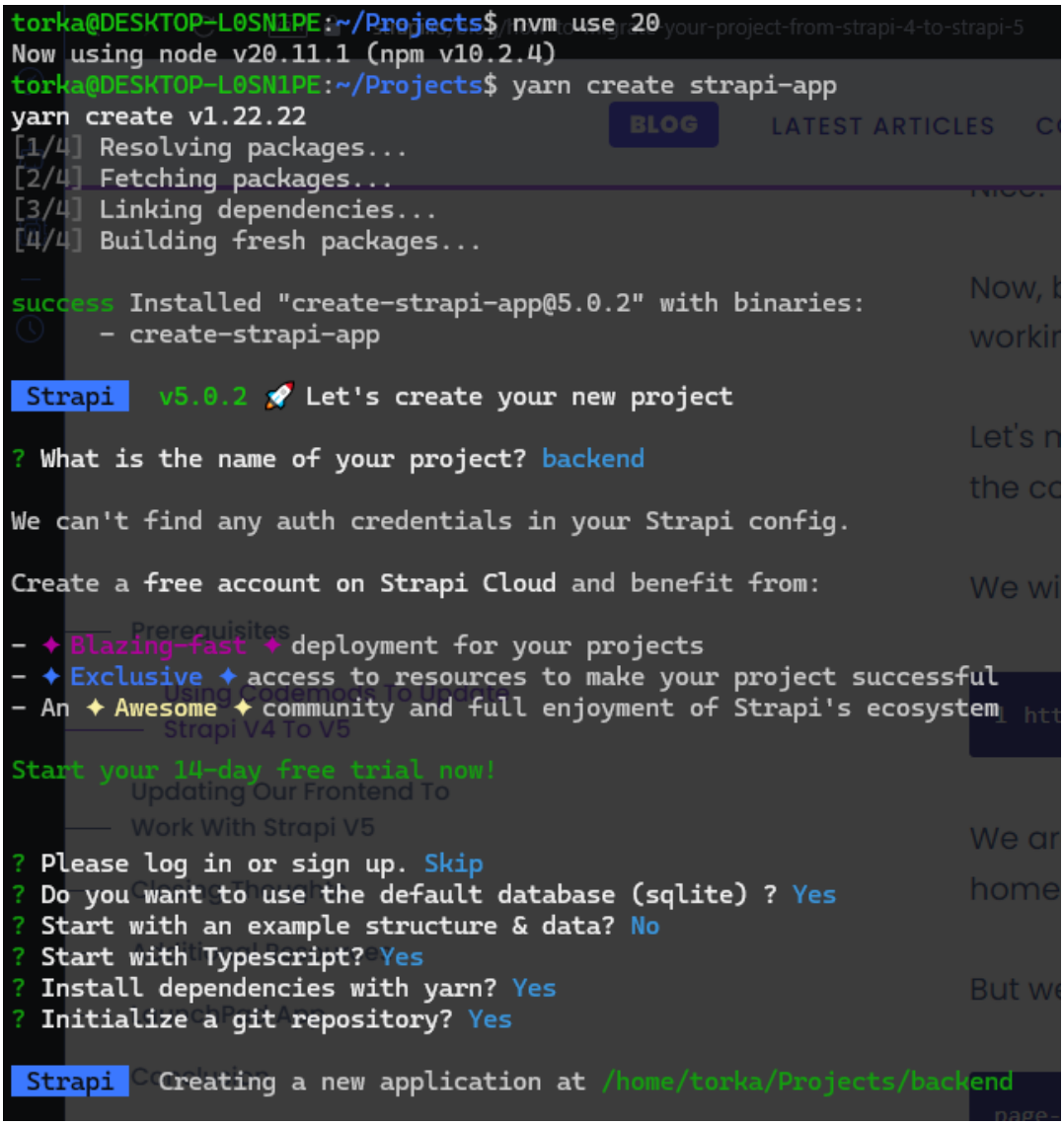


FIGURE 6 – Création d'un projet Strapi avec le CLI

4. **Connexion ou inscription** : `Skip`
   Je ne souhaite pas me connecter ou m'inscrire car je souhaite deployer le projet sur mon VPS et non pas via Strapi Cloud.
5. **Base de données par défaut (sqlite)** : `Yes`
   Pendant le développement, il est pratique d'utiliser une base de données en memoire comme SQLite. En production, on utilisera une base de données plus robuste, PostgreSQL.
6. **Structure et données d'exemple** : `No`
   Je ne veux pas de structure et de données d'exemple.
7. **Utilisation de TypeScript** : `Yes`
   J'utilise TypeScript pour le projet.
8. **Installation des dépendances avec Yarn** : `Yes`
   J'utilise Yarn comme gestionnaire de paquets.

9. **Initialisation d'un dépôt Git** : `Yes`
   J'initialise un dépôt Git pour le projet.

Une fois le projet créé on obtient ce message qui nous indique les principales informations pour gérer le projet :

```
success Saved lockfile.
Done in 96.94s.

      ✓  Dependencies installed

    git  Initializing git repository.

      ✓  Initialized a git repository.

 Strapi  Your application was created!
         Available commands in your project:

         Start Strapi in watch mode. (Changes in Strapi project files will trigger a server restart)
         yarn run develop

         Start Strapi without watch mode.
         yarn run start

         Build Strapi admin panel.
         yarn run build

         Deploy Strapi project.
         yarn run deploy

         Display all available commands.
         yarn run strapi

         To get started run

         cd /home/torka/Projects/backend
         yarn run develop
Done in 135.11s.
torka@DESKTOP-L0SN1PE:~/Projects$
```

FIGURE 7 – Initialisation du projet Strapi terminée

On va donc pouvoir se rendre dans le dossier du projet et lancer le serveur de développement avec la commande `yarn run develop`.

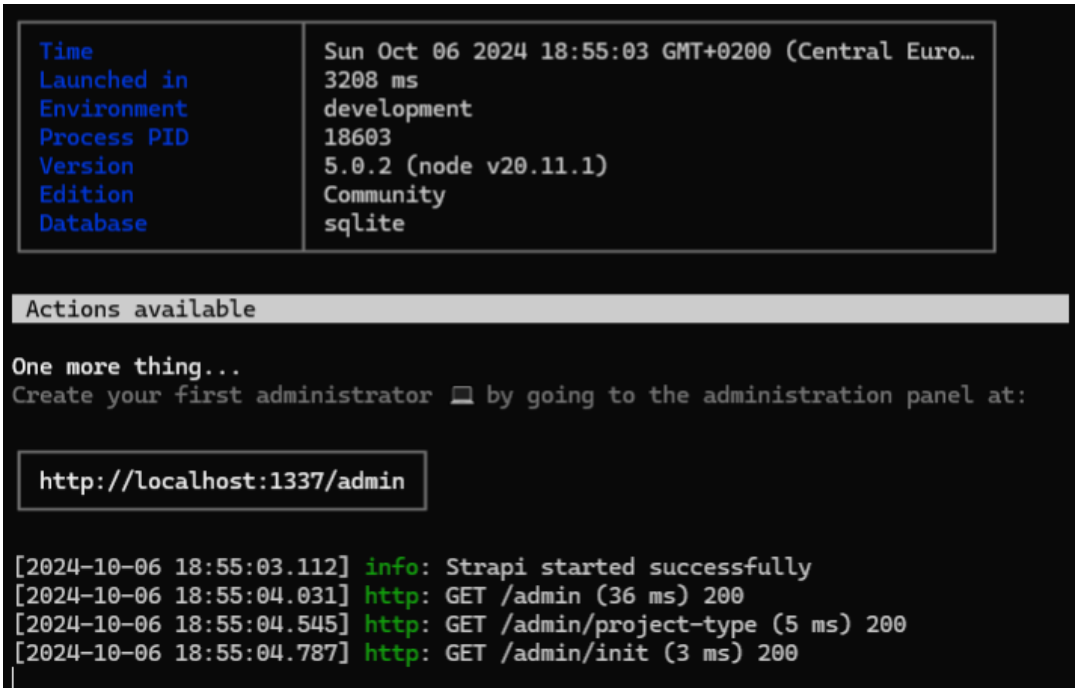


FIGURE 8 – Lancement du serveur de développement

Et voila notre serveur de développement est lancé et notre projet est accessible à l'adresse `http://localhost:1337`. Les variables d'environnement sont générées automatiquement dans le fichier `.env` à la racine du projet. Nous y reviendrons par la suite lors du déploiement du projet. Une fois mon projet initialisé, je fais un premier commit.

5.2.2 *Configuration initiale*

On va donc pouvoir accéder à l'interface d'administration de Strapi à l'adresse `http://localhost:1337/admin`. Il nous est d'abord demandé de créer un utilisateur admin :

Une fois cela fait, nous avons accès à l'interface d'administration de Strapi :

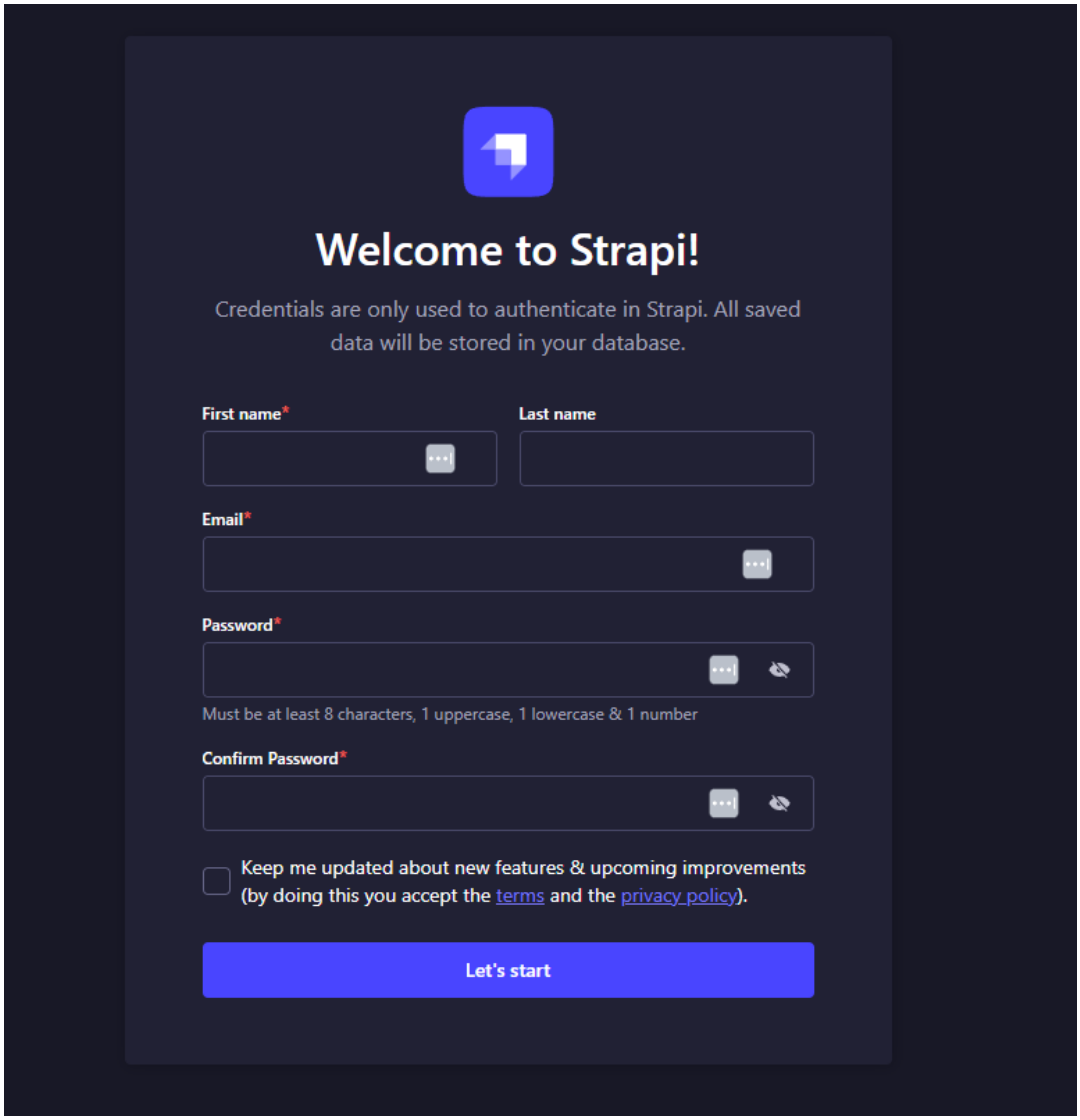


FIGURE 9 – Création de l'utilisateur admin

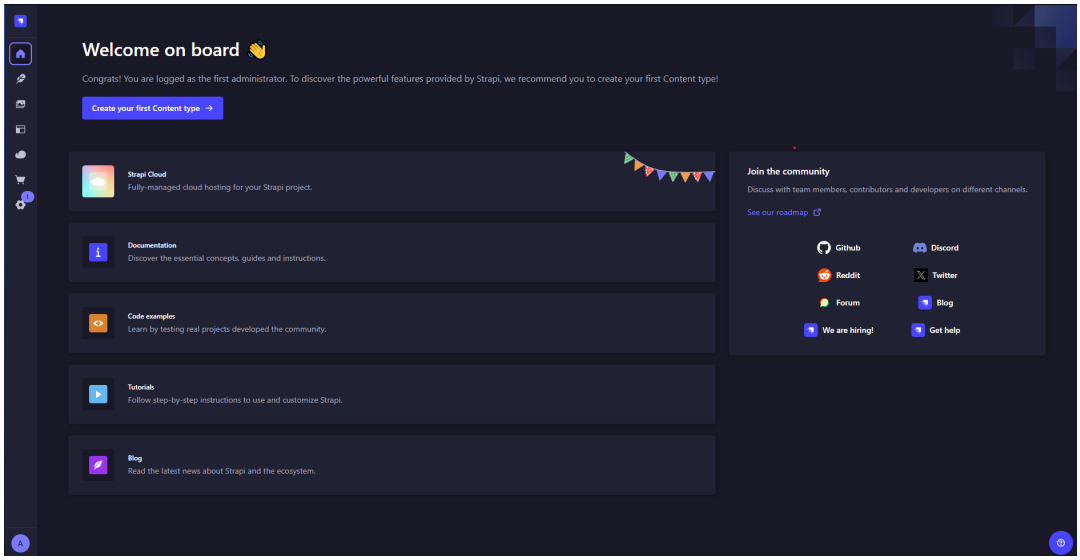


FIGURE 10 – Interface d'administration de Strapi

Il ne sera pas possible de rentrer dans les détails de l'interface afin de ne pas produire un rapport trop long. Mais Strapi offre beaucoup de fonctionnalités intéressantes comme la gestion des medias, des utilisateurs, des permissions, de l'api, etc.

5.3 MODÉLISATION DES DONNÉES

Dans cette section, je vais me contenter de montrer comment fonctionne l'ajout d'une entité. Nous nous concentrerons sur les étapes essentielles pour intégrer une nouvelle entité dans le système, sans entrer dans les détails complexes des autres fonctionnalités.

5.3.1 *Types de Contenu*

Strapi propose trois types principaux de contenus : les Collection Types, les Single Types et les Components.

5.3.1.1 *Collection Types*

Les Collection Types sont utilisés pour les contenus qui peuvent avoir plusieurs entrées. Par exemple, pour notre blog, les articles, les tags, et les médias sont des Collection Types.

— **Articles** : Chaque article est une entrée distincte dans la collection "Articles".
— **Tags** : Chaque tag est une entrée unique dans la collection "Tags".

5.3.1.2 *Single Types*

Les Single Types sont utilisés pour les contenus qui n'ont qu'une seule instance. Dans notre cas, cela inclut :

— **About** : Un seul contenu pour la page "À Propos".
— **BlogHomePage** : Une configuration unique pour la page d'accueil.

5.3.1.3 *Components*

Les Components sont des structures de données réutilisables qui peuvent être incluses dans les Collection Types ou les Single Types. Ils permettent de modulariser et de réutiliser des groupes de champs à travers différents types de contenu.

Dans notre projet de blog, nous pouvons utiliser des Components pour :

— **SEO** : Un component contenant des champs pour les métadonnées SEO (titre, description, mots-clés) qui peut être réutilisé dans les Articles et les Pages.

L'utilisation de Components offre plusieurs avantages :

— **Réutilisabilité** : Les structures de données communes peuvent être facilement partagées entre différents types de contenu.
— **Maintenance simplifiée** : Les modifications apportées à un Component sont automatiquement répercutées partout où il est utilisé.
— **Cohérence** : Les Components assurent une structure cohérente pour des éléments similaires à travers le site.

Pour créer un Component dans Strapi, on utilise également le Content-Type Builder, en choisissant "Create new component" au lieu de "Create new collection type".

5.3.2 *Création d'un Collection Type*

On va donc pouvoir créer notre premier Collection Type. Pour cela on va se rendre dans l'onglet "Content Type Builder" et on va cliquer sur "Create new collection type" (On remarque qu'on a deja type User) :

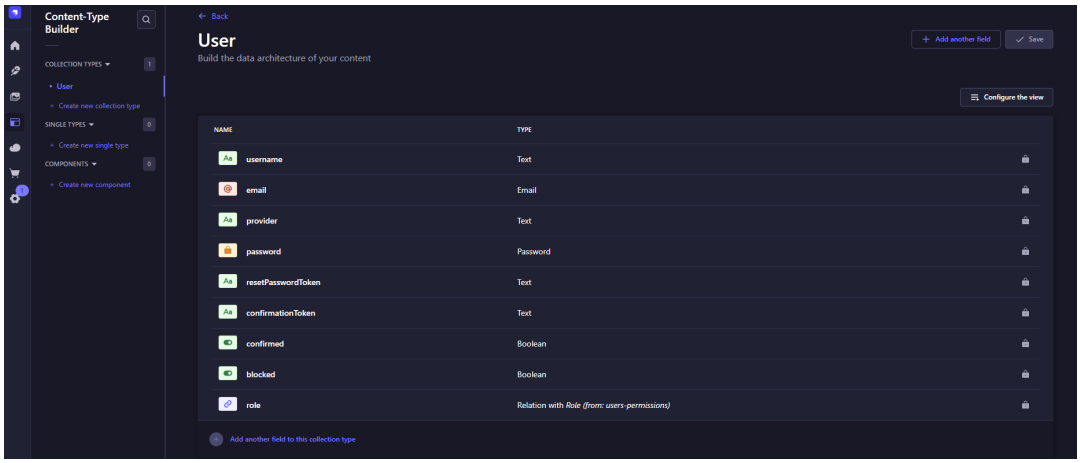


FIGURE 11 – Création d'un Collection Type

On est ensuite invité a saisir le nom de la collection, on va donc saisir "Article". On nous propose egalement de choisir les noms des endpoints pour l'API.

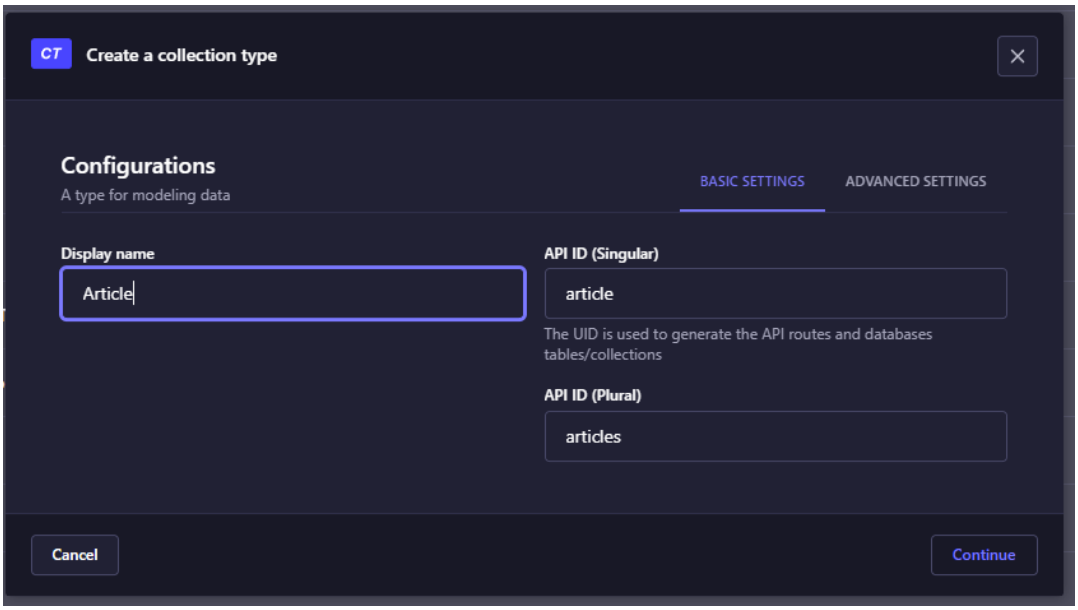


FIGURE 12 – Création d'un Collection Type

Puis on va nous demander de choisir les champs que notre collection doit contenir, un a un. On va donc pouvoir choisir par exemple entre un champ texte, un champ résumé, un champ image, etc.

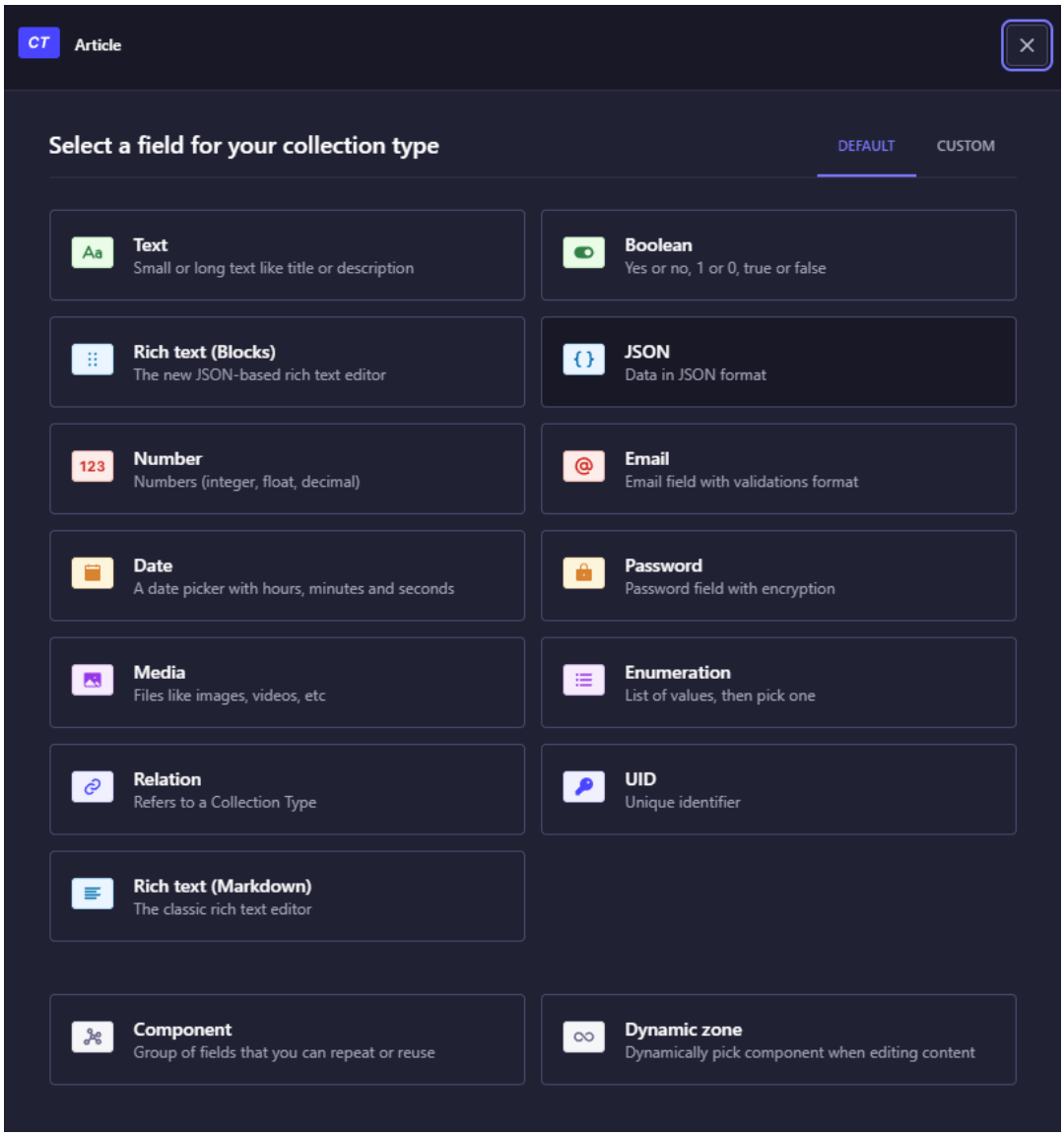


FIGURE 13 – Création des champs pour la collection

Une fois tous les champs voulus renseignés et configurés, on peut cliquer sur "Save". LE serveur de développement va se relancer et creer notre collection. On peut donc voir sa représentation dans l'onglet "Collection Types". Et on peut a présent ajouter un article dans le content manager en faisant create new entry.

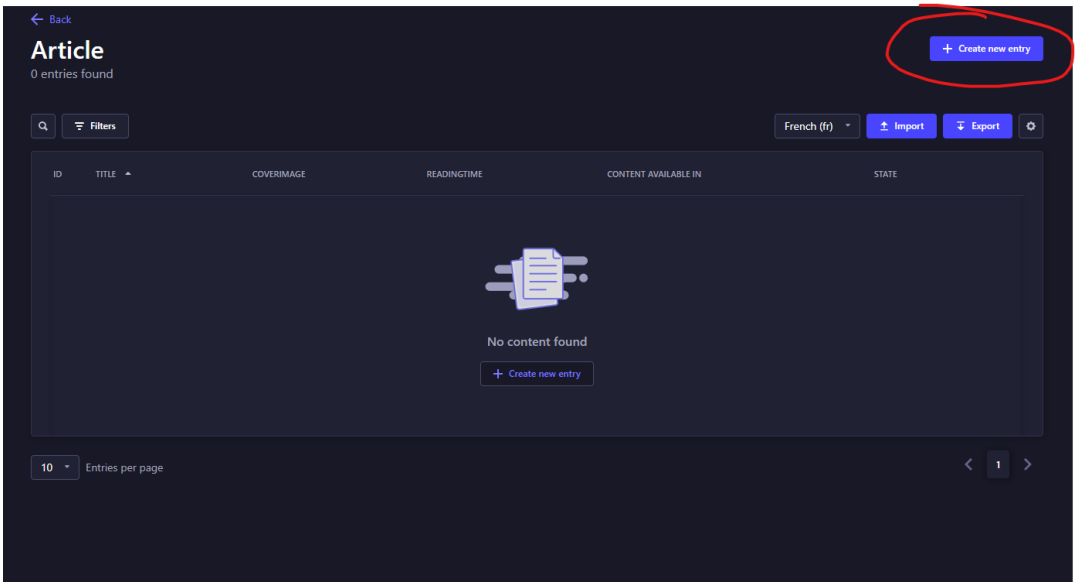


FIGURE 14 – Création de la collection

Tous les champs nous sont demandés, il est possible de définir des champs obligatoire, des contraintes et bien d'autres choses.

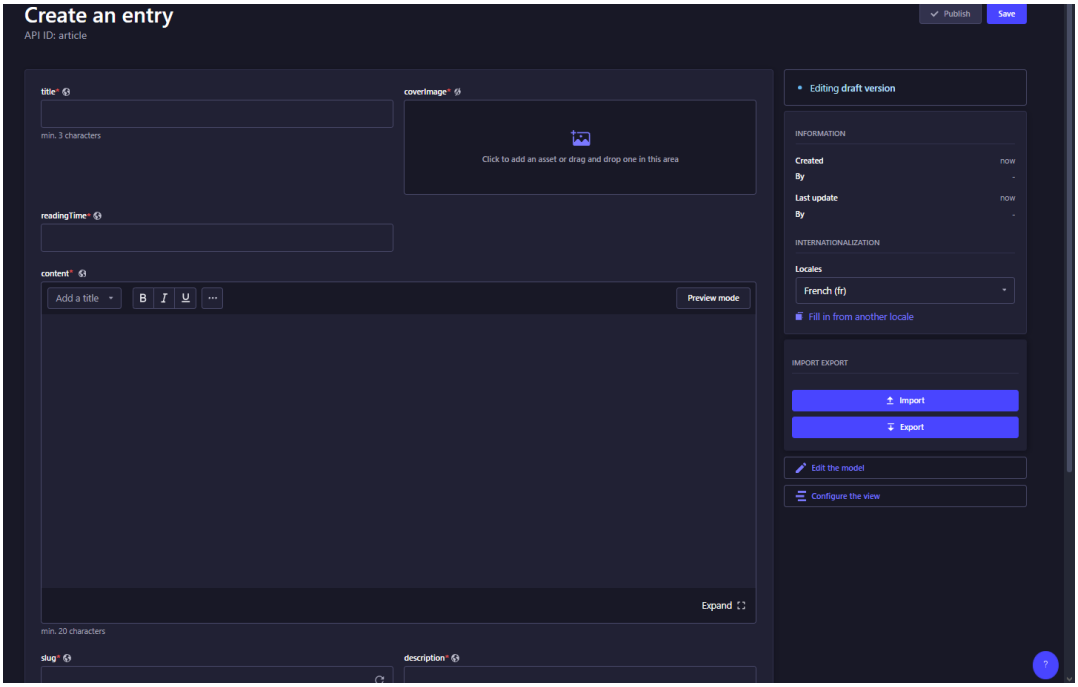


FIGURE 15 – Création de la collection

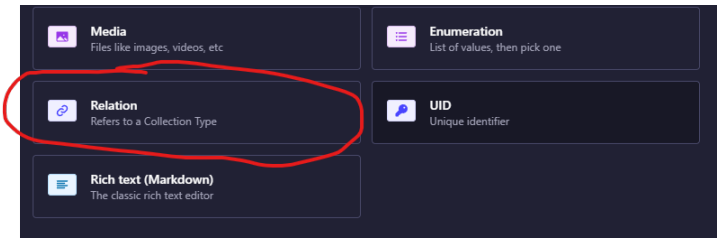
Strapi va ensuite se charger de créer les schémas correspondant pour la création de la table dans notre base de donnée. Tout est accessible et personalisable dans les fichiers du projet, mais je n'aurais pas le temps de couvrir ce point dans le rapport.

A nouveau, on va pouvoir faire un commit du projet pour sauvegarder l'implémentation de notre collection.

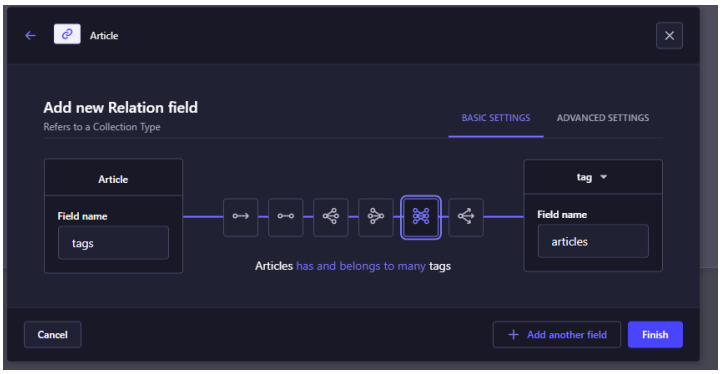
C'est ainsi que j'ai implémenté mon modele entité association, en choisissant les bon type (single, collection et composant) et en ajoutant les champs necessaires.

5.3.3 *Relations entre les Contenus*

Les relations entre les differents contenus sont également gérées à l'aide de champs. On va donc pouvoir ajouter des relations entre les contenus en cliquant sur "Add Relation" et en choisissant le contenus voulu.



(a) Le champ relation.



(b) Les differents types de relations.

FIGURE 16 – L'ajout d'une relation.

5.4 API ET ENDPOINTS

5.4.1 *Configuration de l'API*

Il est possible de configurer les endpoints de l'api, notamment de definir les methodes authorisées (GET, POST, PUT, DELETE) et les permissions. La génération de token d'authentification est également possible. Dans mon cas l'api sera publique, mais il est possible de limiter l'accès à des endpoints spécifiques pour des utilisateurs authentifiés ou non.

5.5 GESTION DES MÉDIAS

Les médias sont gérés de la même manière que les types de contenus. On va donc pouvoir les ajouter, les modifier, les supprimer. Il est possible de les organiser dans des dossiers, de gérer des propriétés utiles comme les captions, le texte alternatif, etc. Strapi a également la possibilité de générer differentes tailles d'images (thumbnail, small, medium, large) pour chaque image ajoutée, le tout disponible depuis l'api.

5.6 CONCLUSION

Voici donc une rapide introduction à Strapi et un aperçu de comment je m'en suis servi pour l'implémenter dans mon projet.

6 FRONTEND AVEC NEXT.JS

6.1 INTRODUCTION À NEXT.JS

Next.js est un framework React populaire qui offre des fonctionnalités avancées telles que le rendu côté serveur (SSR), la génération de sites statiques (SSG), et le routage automatique. Il est particulièrement adapté pour créer des applications web performantes et évolutives. Basé sur React, il permet de créer des composants React réutilisables, et de les combiner pour créer des pages web.

6.2 COMPOSANTS SERVEUR ET COMPOSANTS CLIENT

Next.js 13 introduit une distinction importante entre les composants serveur et les composants client :

6.2.1 *Composants Serveur*

Les composants serveur sont rendus sur le serveur. Ils offrent plusieurs avantages :

- Accès direct à la base de données et aux ressources du serveur
- Meilleure sécurité pour les données sensibles
- Réduction de la taille du bundle JavaScript côté client
- Amélioration des performances de chargement initial

6.2.2 *Composants Client*

Les composants client sont rendus dans le navigateur. Ils sont utiles pour :

- L'interactivité et les événements utilisateur
- L'utilisation des hooks React et du state local
- L'accès aux APIs du navigateur

6.2.3 *Utilisation par défaut*

Il est important de noter que dans Next.js 13 avec App Router, tous les composants sont par défaut des composants serveur. Pour créer un composant client, il faut explicitement l'indiquer en ajoutant la directive "use client" au début du fichier.

Cette approche permet d'optimiser les performances en rendant par défaut les composants sur le serveur, tout en offrant la flexibilité d'utiliser des composants clients lorsque l'interactivité est nécessaire.

6.3 INSTALLATION ET CONFIGURATION

Pour créer un nouveau projet Next.js avec App Router, nous utiliserons le CLI de Next.js :

```
npx create-next-app
```

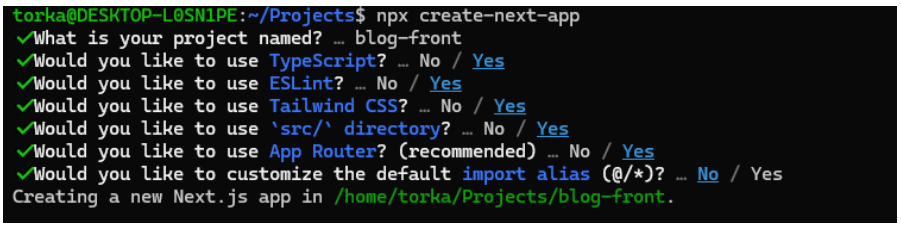


FIGURE 17 – Création d'un projet NextJS

- **Nom du projet** : `blog-front`
  - Le projet sera créé avec ce nom.
- **TypeScript** : `Yes`
  - Le projet utilisera TypeScript, un sur-ensemble de JavaScript qui ajoute des types statiques.
- **ESLint** : `Yes`
  - ESLint sera utilisé pour analyser le code et identifier les problèmes de style et d'erreurs.
- **Tailwind CSS** : `Yes`
  - Tailwind CSS, un framework CSS utilitaire, sera intégré au projet.
- **Répertoire `src/`** : `Yes`
  - Le projet utilisera un répertoire `src/` pour organiser le code source.
- **App Router** : `Yes`
  - L'App Router recommandé sera utilisé pour gérer les routes de l'application.
- **Alias d'importation par défaut** : `No`
  - Les alias d'importation par défaut ne seront pas personnalisés.

Le projet est créé dans le répertoire `/home/torka/Projects/blog-front`.

Il ne reste plus qu'à installer les dépendances (je supprime le package-lock.json afin d'utiliser yarn comme gestionnaire de paquets) :

```
yarn install
```

J'ai ensuite configuré Tailwind CSS et Prettier, et ajouté des plugins Tailwind qui seront utiles pour le développement du projet.

6.4 STRUCTURE D'UN PROJET NEXTJS

Un projet Next.js avec App Router a la structure suivante :
— `/app` : Contient les routes et les composants de l'application
— `/public` : Pour les fichiers statiques
— `/components` : Pour les composants React réutilisables (optionnel)

6.5 ROUTING DANS NEXT.JS AVEC L'APP ROUTER

Next.js utilise un système de routage basé sur le système de fichiers. Avec l'App Router, la structure des dossiers et fichiers détermine les routes de l'application.

6.5.1 *Structure de base*

— **`app/`** : Le dossier racine pour les routes.
  — `page.js` : Définit le contenu de la route. Chaque fichier `page.js` correspond à une route unique.
  — `layout.js` : Définit la mise en page partagée pour les pages dans le même dossier. Il persiste entre les navigations.
  — **Sous-dossiers** : Chaque sous-dossier dans `app/` représente une sous-route.

6.5.2 *Fonctionnement*

— **Pages** : Chaque fichier `page.js` dans le dossier `app/` ou ses sous-dossiers devient une route. Par exemple, `app/about/page.js` correspond à la route `/about`.
— **Layouts** : Les fichiers `layout.js` permettent de partager des composants (comme des en-têtes ou des pieds de page) entre plusieurs pages. Ils sont utiles pour maintenir une cohérence visuelle.
— **Routes Dynamiques** : L'utilisation de crochets permet de créer des routes dynamiques, par exemple `[id].js` pour `/post/1`. Les paramètres dynamiques sont ensuite accessibles dans le composant.
— **Erreurs** : Les fichiers `error.js` permettent de gérer les erreurs de manière globale. En englobant le composant dans un Error Boundary [1].

1. Une Error Boundary est un composant React qui capture les erreurs JavaScript n'importe où dans son arbre de composants enfants, les enregistre, et affiche une

— **Chargement Asynchrone** : Les fichiers `loading.js` permettent de gérer le chargement des données avant le rendu. En englobant le composant dans un Suspense [2].

Ce système simplifie la gestion des routes et la réutilisation des composants dans une application Next.js.

6.6 SSR ET SSG DANS NEXT.JS

Le Server-Side Rendering (SSR) et le Static Site Generation (SSG) sont deux méthodes de rendu utilisées dans Next.js pour générer des pages web.

6.6.1 *Server-Side Rendering (SSR)*

Le SSR génère les pages dynamiquement à chaque requête. Cela signifie que le serveur crée le HTML à la volée, en fonction des données actuelles. Cette méthode est utile pour les pages qui nécessitent des données à jour à chaque visite, comme les tableaux de bord ou les pages personnalisées.

6.6.2 *Static Site Generation (SSG)*

Le SSG, en revanche, génère les pages au moment de la construction du site. Les pages sont pré-rendues en HTML et servies directement aux utilisateurs. Cela offre des temps de chargement plus rapides et une meilleure performance, car le contenu est déjà prêt à être affiché.

6.6.3 *Pourquoi utiliser le SSG*

Je tiens à utiliser le SSG car il offre des avantages significatifs en termes de performance et de scalabilité. Les pages statiques se chargent plus rapidement, ce qui améliore l'expérience utilisateur. De plus, le SSG réduit la charge sur le serveur, car les pages sont servies directement sans nécessiter de traitement supplémentaire. Cela le rend idéal pour les sites avec du contenu qui ne change pas fréquemment, comme les blogs ou les pages d'information. D'un point de vue SEO, le SSG est également plus performant, car les pages sont indexées par les moteurs de recherche plus rapidement.

---

interface de substitution à la place de l'arbre de composants qui a planté. Cela évite à l'application de planter entièrement.

2. Suspense est un composant React qui permet de "suspendre" le rendu d'un composant pendant qu'il attend que quelque chose se produise, comme le chargement de données. Il permet d'afficher un contenu de remplacement (comme un indicateur de chargement) pendant l'attente.

Toutes les pages du projets seront donc générées statiquement. Dans la mesure où les articles et les tags peuvent etre modifiés, j'utiliserai des webhooks strapi pour re-générer les pages concernées quand une modification est faite.

Un webhook est un mécanisme automatisé permettant à une application d'envoyer des notifications en temps réel à d'autres systèmes lorsque certains événements se produisent.

6.7 IMPLÉMENTATION DANS NEXTJS

Dans cette section, je vais présenter l'implémentation des pages et des composants dans mon projet Next.js, en mettant l'accent sur la structure et les choix techniques.

6.7.1 *Structure du dossier app*

Grâce à la sitemap précédemment établie (voir Figure 5), j'ai pu facilement définir les pages de mon projet. Cette structure claire m'a permis d'organiser efficacement l'architecture de mon application Next.js en alignant les routes avec la hiérarchie présentée dans la sitemap. Voici à quoi ressemble mon dossier app :

- **app/** : Dossier racine pour les routes de l'application.
  - **about/**
    - `page.tsx` : Fichier de la page pour la route `/about`.
  - **articles/**
    - **[slug]/**
      - `page.tsx` : Fichier de la page dynamique pour les articles, utilisant un paramètre `slug` dans l'URL.
  - **tags/**
    - **[name]/**
      - `page.tsx` : Fichier de la page dynamique pour les tags, utilisant un paramètre `name` dans l'URL.
  - **webhooks/revalidate/**
    - `route.ts` : Fichier de route pour gérer les webhooks de revalidation, envoyé par strapi.
- `error.tsx` : Page d'erreur personnalisée globale.
- `favicon.ico` : Icône du site.
- `globals.css` : Fichier CSS global pour les styles de l'application.
- `layout.tsx` : Fichier de mise en page globale pour l'application, contient le header et le footer.
- `not-found.tsx` : Page personnalisée pour les routes non trouvées, peut être affichée avec la fontion notFound() dans les composants.
- `page.tsx` : Page d'accueil de l'application.

6.7.2 *Création des Composants*

Une fois les pages définies, je peux passer à la création des composants. J'ai au préalable établi un diagramme des composants de mon

projet (voir Figure 37 dans le chapitre B). Les composants sont stockés dans le dossier `src/components/ui`. Voici l'architecture de ce dossier :

- **components/ui/** : Dossier principal pour les composants d'interface utilisateur.
  - **article-card/**
    - `ArticleCard.tsx` : Composant pour afficher un article.
    - `ArticleCardFallback.tsx` : Composant affiché en cas d'erreur dans le composant `ArticleCard`.
  - **article-list/**
    - `ArticleList.tsx` : Composant pour afficher une liste d'articles.
  - **footer/**
    - `Footer.tsx` : Composant pour le pied de page du site.
  - **header/**
    - `Header.tsx` : Composant pour l'en-tête du site.
    - `ThemeSwitch.tsx` : Composant pour changer le thème de l'application.
  - **tag-card/**
    - `TagCard.tsx` : Composant pour afficher un tag.

Cette structure organise les composants par fonctionnalité, facilitant leur gestion et leur réutilisation dans l'application.

6.7.3 *Les Interfaces Typescript*

Les interfaces Typescript servent à définir les types de données utilisés dans l'application. Elles permettent de s'assurer que les données sont correctes et de faciliter la détection d'erreurs, limitant ainsi de nombreuses erreurs potentielles et améliorant la robustesse du code. Elles permettent egalement la complétion automatique des données dans les IDE comme VSCode. Elles sont stockées dans le dossier `src/interfaces`.

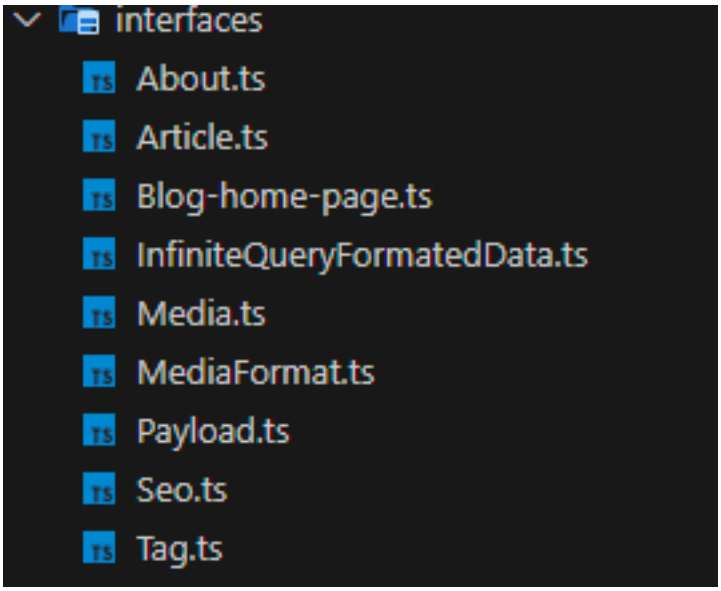


FIGURE 18 – Les Interfaces Typescript

Des exemples d'interfaces sont présentés dans le chapitre B (voir Figure 44 et Figure 45).

6.7.4 *Les Providers*

Les providers sont des composants React qui permettent de gérer les données dans l'application. Elles permettent de fournir un *state* global à l'application. Ils sont stockés dans le dossier `src/providers`.

Un state est un objet qui contient des données qui peuvent changer au fil du temps dans une application React. Il représente l'état actuel de l'application et peut être mis à jour pour refléter les changements d'interface utilisateur.

La structure des providers dans l'application Next.js est la suivante :

- **providers/**
  - `ReactQueryProviderComponent.tsx` : Composant qui configure et fournit le contexte pour React Query[3], facilitant la gestion des requêtes de données asynchrones. Il est utilisé pour implémenter un scroll infini des articles.
  - `ThemeProviderComponent.tsx` : Composant qui gère le thème de l'application, permettant de changer et d'appliquer des styles globaux.

Des exemples de providers sont présentés dans le chapitre B (voir Figure 43 et 42).

6.7.5 *Les fichiers utilitaires*

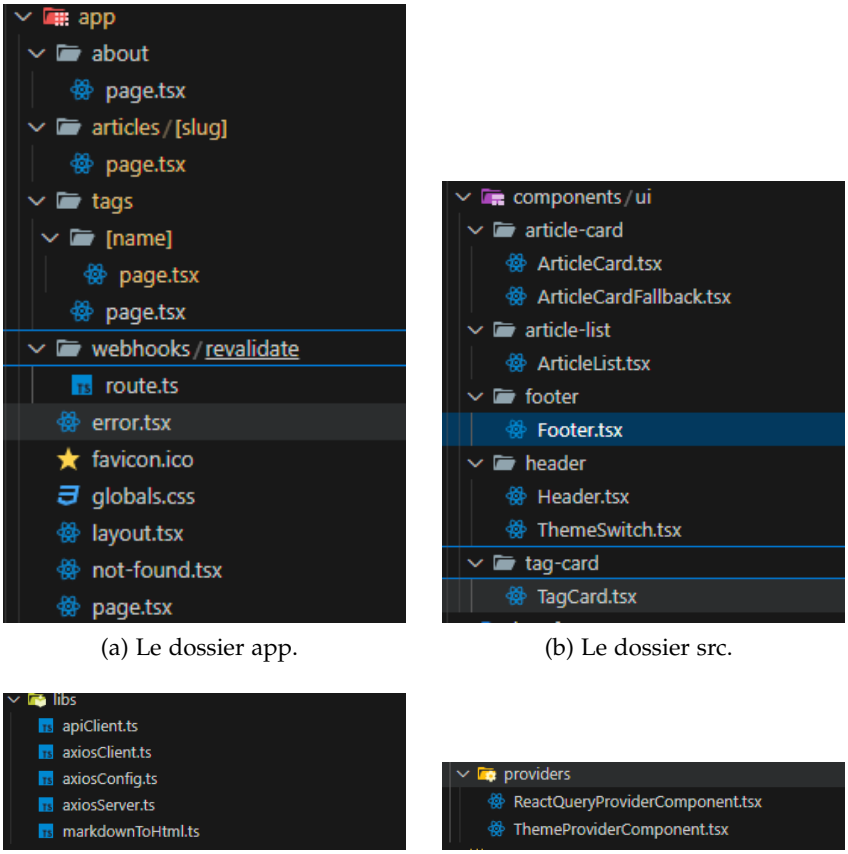
Les fichiers utilitaires sont stockés dans le dossier `src/lib` :

3. React Query est une bibliothèque pour gérer les données asynchrones dans une application React. Elle permet de récupérer, de mettre en cache, de mettre à jour et de supprimer des données de manière efficace, facilitant la gestion des requêtes de données asynchrones.

— **libs/**
  — `axiosConfig.ts` : Fichier qui créé la configuration de axios, notamment pour les headers, la gestions des erreurs avec des interceptors et qui creé un deux instance de client axios, une pour le client et une pour le serveur.
  — `axiosClient.ts` : Les requetes HTTP côté client.
  — `axiosServer.ts` : Les requetes HTTP côté serveur.
  — `markdownToHtml.ts` : Utilitaire pour convertir du contenu Markdown en HTML.

Des exemples de fichiers utilitaires sont présentés dans le chapitre B (voir Figure 38, 39 et 40).

6.7.6 *Capture des Dossiers et Fichiers*



(a) Le dossier app.

(b) Le dossier src.

(c) Le dossier libs.

(d) Le dossier providers.

FIGURE 19 – L'architecture du projet.

6.8 EXEMPLE D'IMPLÉMENTATION (LA PAGE TAG/[NAME])

La page Tags/[name] affichera tous les articles liés à ce tag avec une présentation du tag en question. Cette page servira ici à donner un exemple d'implémentation. Chaque fichier page.tsx doit exporter

un composant React qui sera le composant racine de la page. Je vais dans une première partie présenter le code en amont du composant, puis dans une seconde partie le code du composant.

6.8.1 *Le code en amont*

```tsx
import { fetchTagByName, fetchTags } from "@/libs/axiosServer";
import React from "react";
import Image from "next/image";
import ArticleList from "@/components/ui/article-list/ArticleList";
import { Metadata } from "next/types";
import { notFound } from 'next/navigation';

/**
 * This builtin in next js function allows to generate params so we can render pages on the server at build time and cache them this will improve performances and seo
 *
 */
export async function generateStaticParams() {
  const tagsPayload = await fetchTags();
  const tags = tagsPayload.data;

  return tags.map((tag) => ({
    name: tag.attributes.name,
  }));
}


// This function generates metadata for the tag page
export async function generateMetadata({ params }: { params: { name: string } }): Promise<Metadata> {
  try {
    // Fetch the tag data using the name from the URL parameters
    const tagPayload = await fetchTagByName(params.name);

    // Check if the tag exists
    if (!tagPayload.data || tagPayload.data.length === 0) {
      // If no tag is found, return arbitrary metadata for not found
      return {
        title: 'Tag Not Found',
        description: 'The requested tag could not be found.',
      };
    }

    const tag = tagPayload.data[0];
    const seo = tag.attributes.seo;

    // If there's no SEO data, return basic metadata
    if (!seo) {
      return {
        title: `#${tag.attributes.name}`,
        description: tag.attributes.description,
      };
    }

    // If SEO data exists, return a more comprehensive metadata object
    return {
      // Basic metadata
      title: seo.metaTitle,
      description: seo.metaDescription,
      keywords: seo.keywords,
      robots: seo.metaRobots,
      viewport: seo.metaViewport,
      canonical: seo.canonicalURL,

      // Open Graph metadata for social media sharing
      openGraph: {
        images: seo.metaImage ? [seo.metaImage.data.attributes.url] : [],
      },

      // Twitter-specific metadata
      twitter: {
        card: 'summary_large_image',
        images: seo.metaTwitterImage ? [seo.metaTwitterImage.data.attributes.url] : [],
      },

      // Include structured data if it exists
      ...(seo.structuredData && {
        other: {
          'script:ld+json': JSON.stringify(seo.structuredData),
        },
      }),
    };
  } catch (error) {
    console.error("Error generating metadata for tag:", error);
    // Return arbitrary metadata for error case
    return {
      title: 'Error Loading Tag',
      description: 'There was an error loading the requested tag.',
    };
  }
}


// Generate page on demand if path hasn't been regenerated yet
export const dynamicParams = true;
```

FIGURE 20 – La page Tag/[name] - partie 1

Importations

— `fetchTagByName, fetchTags` : Fonctions pour récupérer des tags depuis le serveur.
— `React, Image, ArticleList, Metadata, notFound` : Importations de React et Next.js pour gérer les composants et la navigation.

Fonction `generateStaticParams`

— Utilisée pour générer des paramètres statiques à la construction du site (SSG).
— Récupère tous les tags et retourne un tableau d'objets avec le nom de chaque tag.
— Améliore les performances et le SEO en pré-rendant les pages au moment du build.

Fonction `generateMetadata`

— Génère des métadonnées pour une page de tag spécifique.
— Récupère le tag en utilisant le nom passé dans les paramètres d'URL.
— Si le tag n'existe pas, retourne des métadonnées par défaut indiquant que le tag n'a pas été trouvé.
— Si le tag existe, vérifie s'il y a des données SEO :
  — Retourne des métadonnées de base si aucune donnée SEO n'est présente.
  — Retourne un objet de métadonnées complet si des données SEO sont disponibles, incluant des informations pour Open Graph et Twitter.
— Gère les erreurs en retournant des métadonnées d'erreur.

Exportation `dynamicParams`

— Définit `dynamicParams` à `true` pour permettre la génération de pages à la demande si le chemin n'a pas encore été régénéré.

Le composant

```tsx
const TagPage = async ({ params }: { params: { name: string } }) => {
  try {
    // Retrieving the tag as a strapi payload
    const tagPayload = await fetchTagByName(params.name);

    // Check if the tag exists
    if (!tagPayload.data || tagPayload.data.length === 0) {
      // If no tag is found, redirect to the 404 page
      notFound();
    }

    // Assigning the tag data to a variable
    const tag = tagPayload.data[0];

    const coverImage = tag.attributes.cover.data;

    return (
      <main className="container flex flex-col gap-8 lg:max-w-screen-lg">
        <div className=" lg:px-10 xl:px-20">
          <h1 className="mb-6 text-2xl font-medium leading-normal text-neutral-900 dark:text-neutral-300 sm:text-4xl sm:leading-10 lg:mb-8 lg:text-5xl xl:text-6xl">
            {`#${tag.attributes.name}`}
          </h1>
          <p className="text-base font-medium text-neutral-700 dark:text-neutral-400 sm:text-lg sm:leading-7 lg:text-xl lg:leading-loose">
            {tag.attributes.description}
          </p>
        </div>
        {/* #TODO Handle image sizing */}
        <div>
          <div className="relative mx-auto  mb-4 h-[340px] w-full lg:h-[500px] ">
            <Image
              // Image url structure is : strapiUrl/uplooads/image
              src={`${process.env.STRAPI_URL}${coverImage.attributes.url}`}
              alt={`${coverImage.attributes.alternativeText}` || "cover image"}
              fill
              className=" rounded-xl object-cover"
            />
          </div>
          <p className="text-center text-xs font-normal italic leading-none text-neutral-600 dark:text-neutral-500">
            {coverImage.attributes.caption}
          </p>
        </div>
        <div className=" lg:px-10 xl:px-20">
          <ArticleList
            queryParams={`filters[tags][name][$in]=${tag.attributes.name}`}
          />
        </div>
      </main>
    );
  } catch (error) {
    console.error("Error fetching tag:", error);
    notFound();
  }
};

export default TagPage;
```

FIGURE 21 – La page Tag/[name] - aprite 2

Fonction `TagPage`

— **Paramètres** : Accepte un objet `params` (comme le requiert Next.js) avec un `name` pour identifier le tag. Le `name` est passé par le générateur de page statique.
— **Récupération des données** : Utilise `fetchTagByName` pour obtenir les données du tag depuis le serveur Strapi.
— **Vérification de l'existence du tag** :
  — Si le tag n'existe pas ou si les données sont vides, redirige vers une page 404 avec `notFound()`.

Affichage du contenu

- **Variables** :
  - `tag` : Contient les données du premier tag récupéré.
  - `coverImage` : Contient les données de l'image de couverture du tag.
- **Structure HTML** :
  - Utilise des classes Tailwind CSS pour le style.
  - Affiche le nom et la description du tag.
  - Affiche l'image de couverture avec une URL construite à partir de variables d'environnement.
  - Affiche une légende sous l'image.

Composant `ArticleList`

- Utilisé pour afficher une liste d'articles associés au tag, en passant des paramètres de requête pour filtrer par nom de tag.

Gestion des erreurs

- En cas d'erreur lors de la récupération des données, affiche un message d'erreur dans la console et redirige vers une page 404.

Exportation

- Le composant `TagPage` est exporté par défaut pour être utilisé dans l'application comme le requiert Next.js.

6.8.2 *La gestion des erreurs*

6.8.3 *Au niveau des composants*

Dans mon application Next.js, chaque composant est encapsulé dans un *Error Boundary*. Cette approche permet de gérer les erreurs de manière plus élégante et de garantir que l'application reste fonctionnelle même en cas de problème dans un composant spécifique. Cela me permet d'afficher une interface alternative en cas d'erreur.

FIGURE 22 – L'affichage d'une erreur dans le composant ArticleList

6.8.4 *Au niveau des requêtes*

J'ai utilisé axios avec des interceptors pour gérer les erreurs de requêtes (voir Figure 38). Chaque requête retourne une réponse ou une erreur, que je gère dans les composants en fonction de ce que j'attends de la requête. L'utilisation d'interceptors est très pratique pour gérer les erreurs de requêtes, permettant de les gérer de manière centralisée et de ne pas avoir à les gérer dans chaque composant.

6.8.4.1 *Au niveau des pages*

Next.js propose un mécanisme de gestion des erreurs de pages avec la fonction `notFound()` et `redirect()`. J'ai utilisé ces fonctions dans les pages en fonction de ce que j'attendais de la requête.

6.9 REVALIDATION DES PAGES AVEC WEBHOOKS

Next.js permet également de revalider les pages automatiquement à intervalles réguliers. Cependant, j'ai choisi une approche basée sur les webhooks pour économiser les ressources de mon serveur en ne revalidant que lorsque c'est nécessaire.

Dans notre approche de génération de site statique (SSG), il est crucial de maintenir le contenu à jour lorsque des modifications sont apportées dans le CMS Strapi. Pour ce faire, j'ai implémenté un système de revalidation basé sur des webhooks.

6.9.1 *Principe de Fonctionnement*

Lorsqu'un contenu est modifié dans Strapi (création, mise à jour ou suppression d'un article ou d'un tag), un webhook est déclenché. Ce webhook envoie une requête à un point d'entrée spécifique de notre application Next.js, qui à son tour déclenche la régénération des pages statiques concernées.

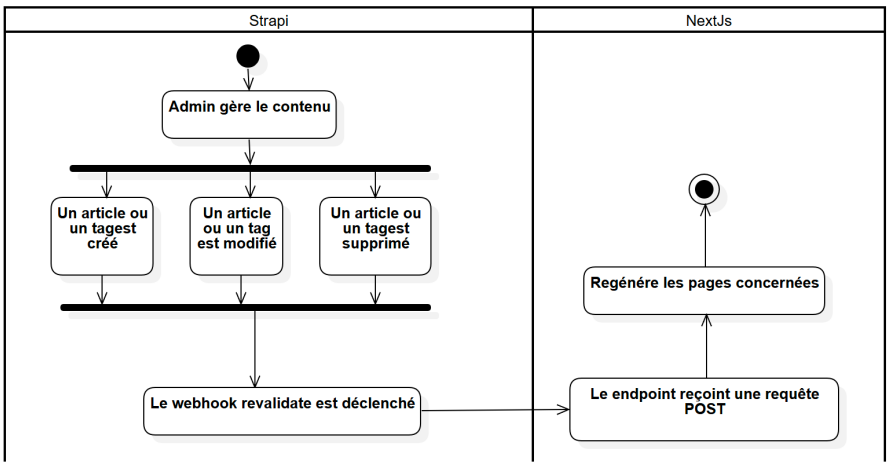


FIGURE 23 – Le diagramme d'activité de la revalidation

6.9.2 *Implémentation dans Next.js*

J'ai créé un point d'entrée API dans le dossier `app/webhooks/revalidate/route.ts` pour gérer les requêtes de revalidation. Ce fichier contient une fonction qui vérifie le type de contenu modifié (article ou tag) et revalide les chemins appropriés en utilisant la fonction `revalidatePath` de Next.js. Une verification de l'origine de la requête est également effectuée à l'aide d'un token, afin de sécuriser le endpoint.

6.9.3 *Configuration dans Strapi*

Dans Strapi, j'ai configuré un webhook pour envoyer une requête POST à l'URL de notre point d'entrée de revalidation chaque fois qu'un article ou un tag est modifié. Le payload[4] du webhook inclut les informations nécessaires pour identifier le type de contenu modifié.

6.9.4 *Avantages de cette Approche*

Cette méthode de revalidation offre plusieurs avantages :
— Maintien de la performance du SSG tout en assurant la fraîcheur du contenu.
— Régénération ciblée des pages, évitant une reconstruction complète du site.
— Automatisation du processus de mise à jour, réduisant la charge de travail manuelle.

Cette implémentation permet de combiner efficacement les avantages du SSG (performance, SEO) avec la flexibilité d'un contenu dynamique géré via un CMS headless comme Strapi. Les détails techniques et le code source complet de cette fonctionnalité peuvent être consultés dans les annexes de ce document.

6.10 LES TESTS

Pour assurer la qualité et la fiabilité de l'application frontend, j'ai mis en place une suite de tests utilisant Jest et React Testing Library. Ces tests sont situés dans le dossier `__tests__` du projet.

J'ai implémenté des tests pour les composants individuels ainsi que pour les pages complètes, en utilisant une combinaison de Jest et React Testing Library. Cette approche permet de vérifier à la fois le comportement des composants isolés et leur intégration dans le contexte plus large de l'application. Bien que la couverture de test ne

4. Le payload, ou charge utile en français, désigne les données transmises dans le corps d'une requête HTTP. Dans le contexte d'un webhook, il contient généralement les informations pertinentes liées à l'événement qui a déclenché le webhook.

soit pas encore complète, elle est en cours d'expansion pour couvrir l'ensemble des composants et des fonctionnalités.

Ces tests jouent un rôle crucial dans la pipeline de déploiement que j'ai mise en place, assurant ainsi la qualité du code à chaque étape du processus de développement et de déploiement.

Pour une analyse plus détaillée, les plans de test complets ainsi que le code source des tests peuvent être consultés dans les annexes de ce document. (Voir Annexe C)

Quatrième partie

DÉPLOIEMENT

Cette partie se concentre sur le déploiement du blog, couvrant les aspects essentiels tels que la mise en production, la configuration des serveurs, et les stratégies pour assurer la performance et la sécurité du site en ligne. Elle aborde également les considérations post-déploiement, comme la maintenance continue et l'optimisation pour garantir une expérience utilisateur optimale.

7 DÉPLOIEMENT

7.1 INTRODUCTION

7.2 CHOIX DE L'INFRASTRUCTURE

Pour le déploiement de mon blog, j'ai soigneusement évalué différentes options d'hébergement en tenant compte de plusieurs facteurs clés tels que les performances, la flexibilité, la scalabilité et le rapport qualité-prix. Après une analyse approfondie, j'ai opté pour un VPS (Serveur Privé Virtuel) proposé par Contabo.

7.2.1 *Caractéristiques du VPS Choisi*

J'ai sélectionné l'offre VPS "No Setup" de Contabo à 5 euros par mois, qui offre les spécifications suivantes :

- **Stockage :** 100 GB
- **Processeur :** 4 vCPU Cores
- **Mémoire RAM :** 6 GB
- **Bande passante :** 32 TB de trafic sortant
- **OS** Ubuntu 20.04

Ces caractéristiques sont largement suffisantes pour héberger mon blog. Et l'option no setup me permet de configurer le serveur comme bon me semble.

7.2.2 *Configuration du VPS*

J'accède au VPS avec SSH[1] et les identifiants fournis par Contabo, je l'ai ensuite entièrement configuré pour répondre à mes besoins, les principales étapes ont été les suivantes :

1. Mise à jour du système
2. Création d'un nouvel utilisateur et désactivation du root login
3. Mise en place de l'authentification par clé SSH
4. Mise en place d'un pare-feu avec UFW
5. Installation de Fail2Ban pour la sécurité
6. Changement du port SSH
7. Mise en place d'un fichier SWAP

1. SSH (Secure Shell) est un protocole de communication sécurisé permettant d'accéder à distance à un serveur et d'y exécuter des commandes.

8. Mise en place de logwatch pour la surveillance et l'envoi de logs quotidiens par mail
9. Installation du Docker Engine

7.3 PROCESSUS DE DÉPLOIEMENT

7.3.1 *Creation des Images Docker*

La première étape consiste à créer les images Docker de production de mon Strapi et de l'application Next.js qui seront utilisé dans le Docker Compose. Les autres images seront directement prises depuis le Docker Hub.

J'ai donc créé un fichier `Dockerfile` dans le dossier de chaque application ((Voir Figure 52) et 53) pour la transformer en image Docker que j'ai ensuite push sur mon Github Registry. Chaque image a été liée au repository qui lui correspondait dans le but de mettre en place une pipeline de deploiement continue.

Les images finales ont été construites en utilisant le principe de multi-stage build pour optimiser la taille de l'image. [2]

7.3.2 *Préparation de l'Environnement de Production*

Une fois les images sur le registry, j'ai pu configurer le fichier `docker-compose.yml` (Voir Figure 55) pour qu'il récupère les images de mon registry, et les déployer sur mon VPS.

Il a également fallu créer l'arborescence de dossiers sur mon VPS pour accueillir les containers. voici à quoi ressemble l'arborescence :

Il n'est pas nécessaire de créer manuellement tous ces dossiers et fichiers. Seuls les fichiers contenant des informations essentielles pour le bon fonctionnement des containers, comme les fichiers .env, doivent être créés à l'avance.

Listing 1 – Arborescence des dossiers

```
monSite/
|-- .env
|-- certbot
|-- nginx
|-- strapi
'-- blogFront/
    '-- .env
```

Les fichiers `.env` contiennent les variables d'environnement nécessaires pour le bon fonctionnement des containers. Une est à la racine du projet et l'autre dans le dossier de l'application Next.js. En effet, Strapi et PostgreSQL partagent des variables d'environnement. Le container Next.js quant à lui a les siennes. Docker créera les dossiers bindés nécessaires pour le bon fonctionnement des containers.

2. Le multi-stage build permet de créer des images Docker plus légères en utilisant plusieurs étapes de construction. La première étape compile le code, tandis que la seconde ne copie que les fichiers nécessaires, réduisant ainsi la taille finale de l'image.

7.3.3 *Stratégie de Déploiement*

Comme indiqué précédemment, j'utiliserai Docker Compose pour déployer mes images. Il me permet de définir et de lancer plusieurs containers en une seule commande, d'après un fichier de configuration établi au préalable. Fichier dans lequel j'ai défini les images à déployer et leurs configurations, l'ordre de démarrage, les volumes pour la persistance des données, etc.

Une fois cela fait, il me suffit de faire un `docker compose up -d` pour déployer mes images et demarrer les containers en arrière plan. Voici ce qui se passe dans les grandes lignes :

1. Docker Compose récupère les images depuis le GitHub Registry si celles présentes sur le VPS ne sont pas à jour.
2. Il crée les containers grâce aux images et aux configurations définit dans le fichier `docker-compose.yml`. Si les containers existent déjà, il les réutilisera.
3. Les containers sont démarrés dans l'ordre indiqué dans le fichier `docker-compose.yml` (Strapi a besoin de PostgreSQL, Next.js a besoin de Strapi, etc.)

7.3.3.1 *Le container reverse-proxy(Nginx)*

Nginx a été configuré pour rediriger les requêtes entrantes vers le bon container, en fonction du domaine. Il écoute les requêtes sur le port 80(http) et 443(https) pour chaque sous domaine. La liste des sous domaines gérés par Nginx est la suivante :

— `aniss.dev` : Le domaine principal, accueillera mon portfolio, Nginx sert actuellement une page statique indiquant que le site est en construction.
— `www.aniss.dev` : Un alias pour `aniss.dev`, Nginx le redirige vers le domaine principal.
— `blog.aniss.dev` : Le sous domaine blog qui accueille mon blog personnel.
— `strapi.aniss.dev` : Le sous domaine qui pointe vers Strapi (Accès à l'API et au portail admin).
— `adminer.aniss.dev` : Permet d'accéder à l'interface d'administration de PostgreSQL.

Nginx force la connexion par HTTPS pour tous les sous domaines en redirigeant les requêtes HTTP vers HTTPS. Il redirige également mon adresse IP vers mon nom de domaine principal.

7.3.3.2 *Le container Certbot*

Certbot fonctionne en collaboration avec le container Nginx pour la gestion automatique des certificats SSL, délivrés par Let's Encrypt.

Cela se fait par le biais d'un challenge HTTP qui vérifie que je possède bien le domaine. L'automatisation est possible en partageant les fichiers nécessaires pour le challenge entre les containers.

7.3.3.3 *Le container StrapiDB(PostgreSQL)*

Le container PostreSQL est utilisé par Strapi pour stocker les données de la base de données. Il utilise un volume Docker pour persister les données, afin de limiter l'accès directe aux données nécessaires à son bon fonctionnement.

7.3.3.4 *Le container Strapi*

Le container Strapi est utilisé pour gérer le contenu de mon blog. Il utilise un dossier bindé pour accéder aux Médias uploadés car ceux-ci ne sont pas stockés dans la base de données. En procédant ainsi, je m'assure de la persistance de ces médias. Divers fichiers de configurations sont bindés afin de permettre de les modifier sans reconstruire l'image.

7.3.3.5 *Le Container blogFront (Next.js)*

L'image a été construite sur un build de production de l'application Next.js, avec l'option standalone. Cette image ne contient pas le code source de l'application mais uniquement le code compilé, les assets et les dépendances nécessaires pour le déploiement. Cela a permis de réduire la taille de l'image (de 2.5Gb à moins de 200Mb) et de rendre le déploiement plus rapide, en effet cette partie de l'application est celle qui est la plus amenée à évoluer.

C'est d'ailleurs pour cette raison que j'ai mis en place une pipeline de déploiement continue.

8

LE MONITORING ET LES TESTS DE CHARGE

8.1 LE MONITORING

Il est crucial de pouvoir monitorer mon environnement de production, que ce soit les containers ou le système d'exploitation. J'ai donc mis en place un dashboard de monitoring avec Prometheus, Grafana et Node Exporter.

— **Prometheus** : C'est un système de surveillance et d'alerte open-source. Son rôle principal est de collecter et stocker des métriques sous forme de séries temporelles. Prometheus interroge régulièrement des endpoints (appelés exportateurs) pour récupérer ces métriques.

— **cAdvisor (Container Advisor)** : Cet outil, développé par Google, est spécifiquement conçu pour collecter, agréger et exporter des informations sur les conteneurs en cours d'exécution. Il fournit des métriques détaillées sur l'utilisation des ressources (CPU, mémoire, réseau, stockage) pour chaque conteneur.

— **Node Exporter** : C'est un exportateur Prometheus qui collecte une large gamme de métriques liées au système d'exploitation. Il fournit des informations sur l'utilisation du CPU, de la mémoire, du disque, du réseau et d'autres métriques au niveau du système, permettant ainsi de surveiller la santé globale du serveur hôte.

— **Grafana Cloud** : C'est une plateforme de visualisation et d'analyse de données hébergée dans le cloud. Elle permet de créer des tableaux de bord interactifs et personnalisables à partir des métriques collectées par Prometheus. Grafana Cloud offre également des fonctionnalités avancées telles que les alertes, l'analyse des logs et le traçage distribué.

L'utilisation combinée de ces outils me permet d'avoir une vue d'ensemble complète de mon infrastructure, depuis les métriques système jusqu'aux performances individuelles des conteneurs, le tout visualisé de manière claire et intuitive via Grafana Cloud.

FIGURE 24 – Dashboard Monitoring Conteneurs

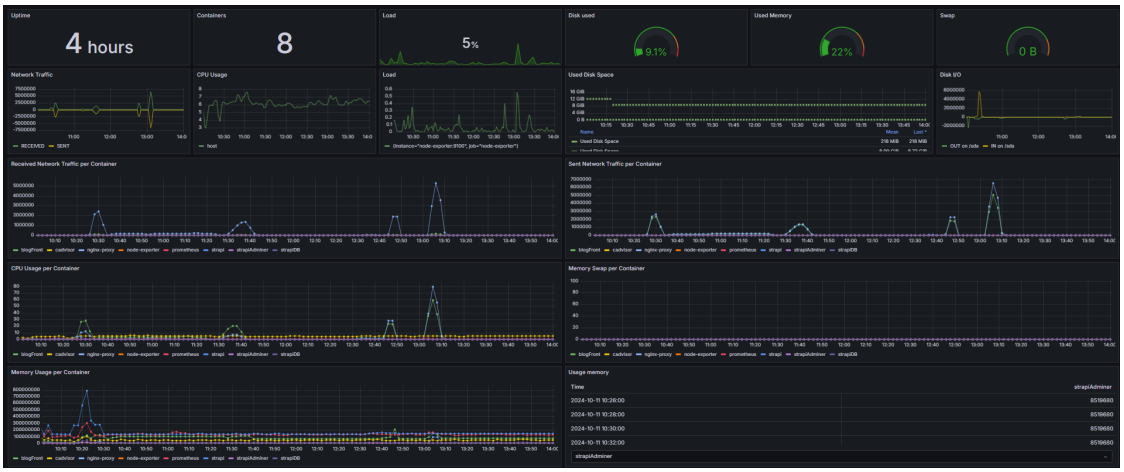


FIGURE 25 – Dashboard Monitoring Host

8.2 LES TESTS DE CHARGE

A présent que mon infrastructure est en ligne et que le monitoring est en place, je peux lancer des tests de charge pour évaluer les performances de mon infrastructure. Il est crucial d'effectuer des tests de charge, d'autant plus sur un environnement qui a été entièrement construit afin de s'assurer qu'il répond correctement aux différentes sollicitations, pour récupérer des données et anticiper les éventuels problèmes futurs.

Pour ce faire, j'ai mis en place 5 plans de tests qui augmentent petit à petit la charge sur les serveurs. En parallèle je me sers de Grafana Cloud pour observer l'évolution des métriques au cours de ces tests. Les tests ont été réalisé avec K6.

1. **Test 1** : Test de charge de base
2. **Test 2** : Test d'Endurance

3. **Test 3** : Test de Montée en Charge avec Pic

Les détails des plans de tests et les résultats complets sont disponibles dans la section C.4.

En somme, le déploiement a été un succès, mon infrastructure répond correctement aux sollicitations et les tests ont montré que celle-ci est capable de supporter une charge importante (plusieurs centaines de requêtes par seconde et plusieurs milliers d'utilisateurs simultanés).

8.3 TEST DE PERFORMANCE

Des tests de performance ont été réalisé avec Lighthouse et la version live du site et ils sont concluants :

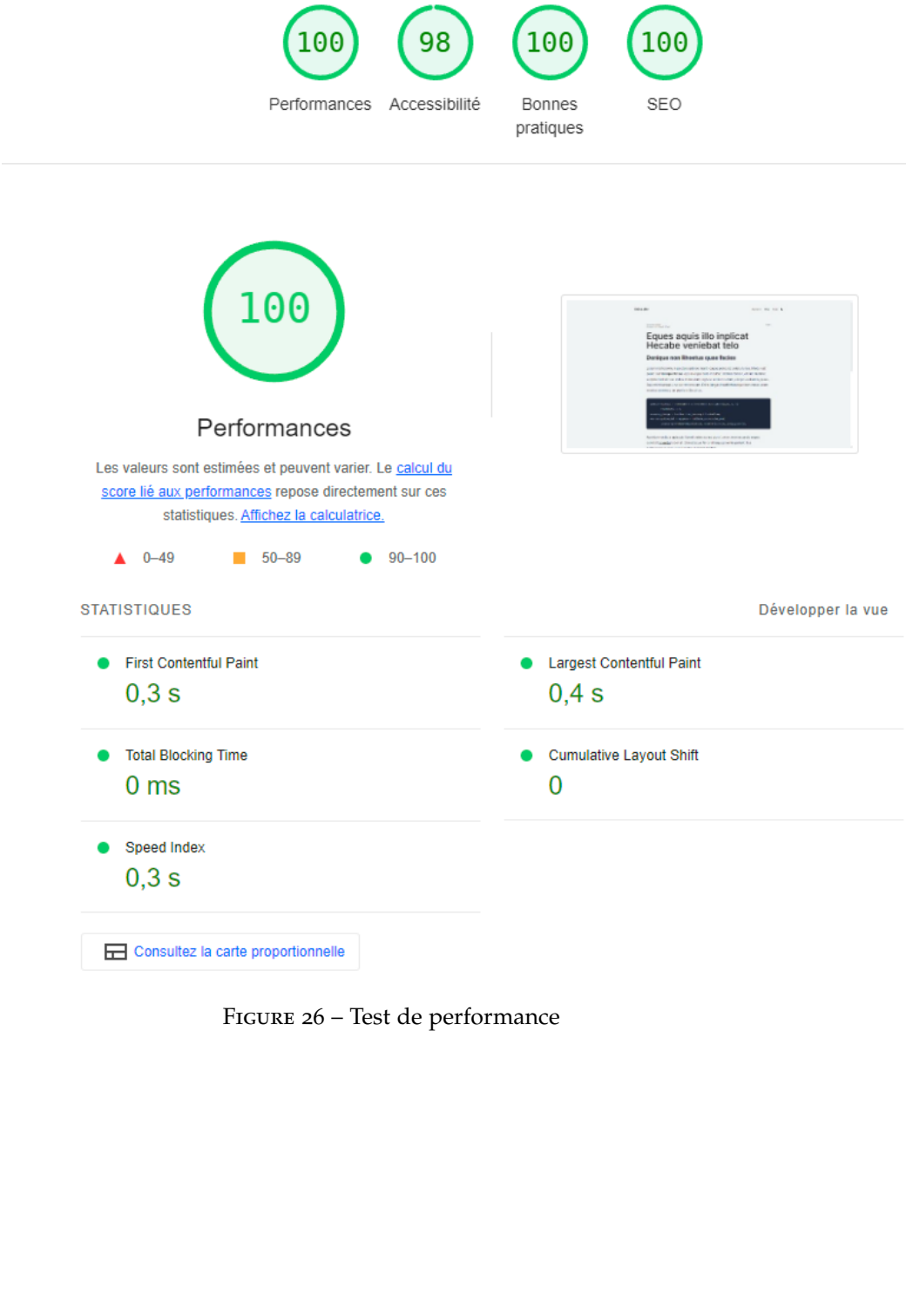


FIGURE 26 – Test de performance

9 PIPELINE DE DEPLOIEMENT CONTINU

9.1 DESCRIPTION DE LA PIPELINE

Afin de fluidifier le processus de développement, de production et de déploiement de mon application, j'ai mis en place une pipeline de déploiement continue pour l'application Next.js. Cette pipeline a été mise en place à l'aide de GitHub Actions.

Elle se décompose en 3 étapes :

1. **Push sur Github** : Un push sur la branche main déclenche une action Github qui récupère les sources de l'application et les compile.
2. **Lancement des tests** : Les tests sont lancés pour s'assurer que le build est bon.
3. **Build et push sur le registry** : L'image est construite et push sur le registry.
4. **Déploiement** : GitHub actions déploie l'image sur le VPS.

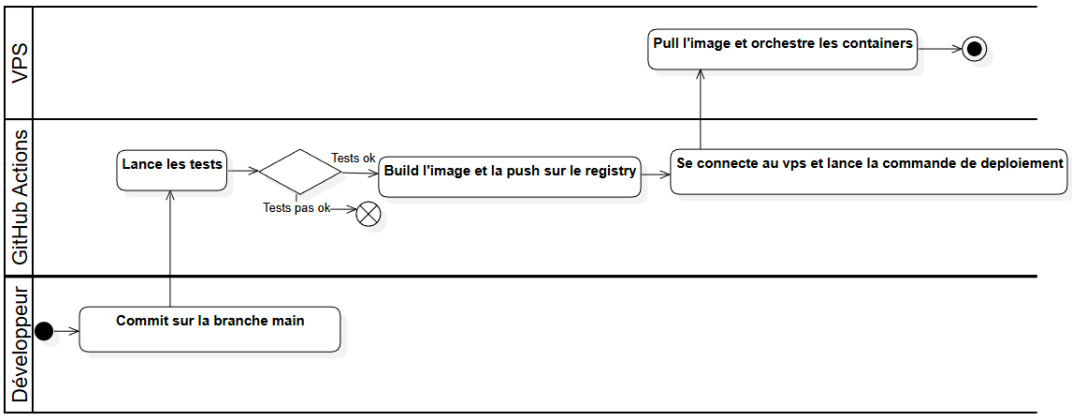


FIGURE 27 – Pipeline de Déploiement Continu

Cela me permet de me concentrer sur le développement sans me soucier de la mise en production. Tout est automatisé et déclenché par un push sur Github. J'ai ensuite accès au détail de chaque action et de chaque étape de la pipeline à l'aide de l'interface Github Actions.

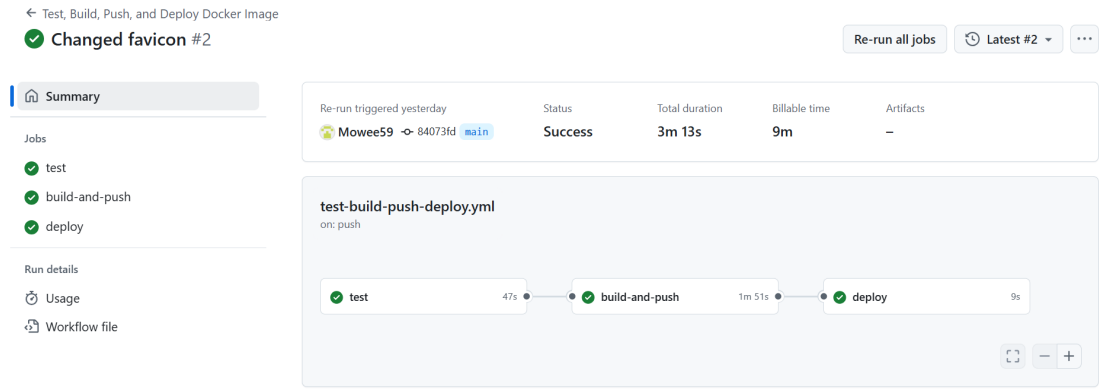


FIGURE 28 – Liste des Workflow Runs

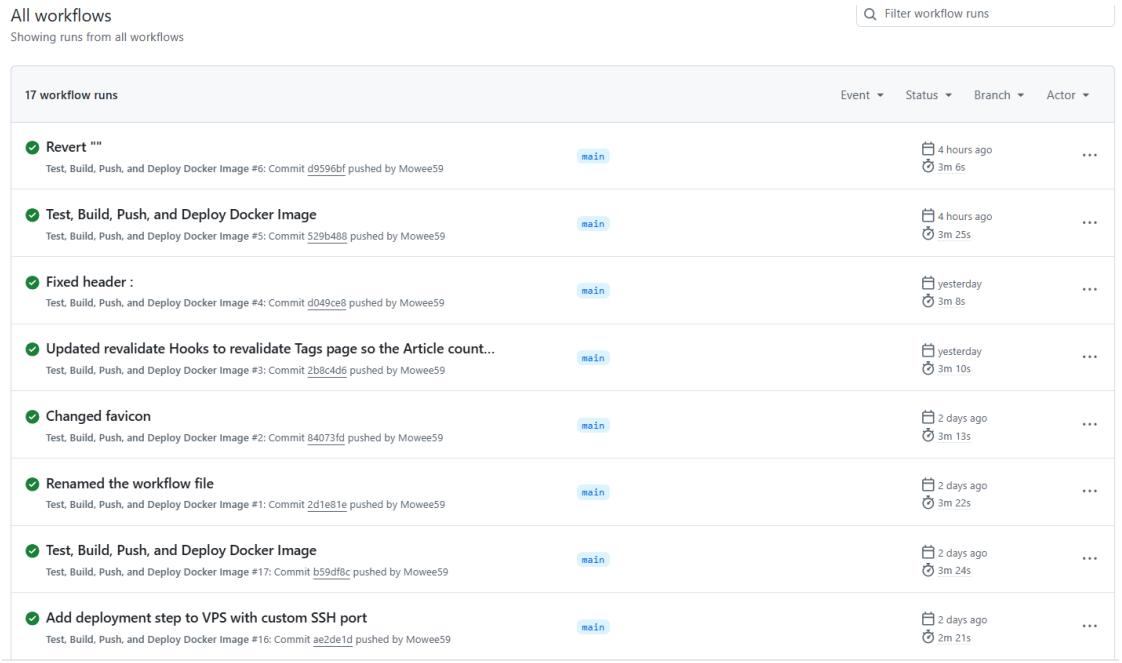


FIGURE 29 – Détail d'un Workflow Run

Ces figures montrent l'interface de GitHub Actions, où l'on peut suivre en détail chaque étape de la pipeline de déploiement. On peut voir dans un premier temps la liste des workflows run, et pour chaque run, les différentes étapes qui sont exécutées (jobs). Il est possible de voir le détail de chaque étape, avec des liens vers les logs.

9.2 LA MISE EN PLACE DU WORKFLOW GITHUB ACTION

Pour mettre en place cette pipeline de déploiement continu, j'ai créé un fichier de workflow GitHub Actions. Ce fichier, nommé "Test, Build, Push, and Deploy Docker Image", définit l'ensemble du processus automatisé. Voici un aperçu des principales étapes du workflow :

1. **Déclenchement** : Le workflow est déclenché à chaque push sur la branche principale (main).
2. **Tests** : Exécution des tests de l'application pour s'assurer de son bon fonctionnement.
3. **Build et Push** : Construction de l'image Docker de l'application et push vers le GitHub Container Registry (ghcr.io).
4. **Déploiement** : Connexion au VPS via SSH et mise à jour des containers avec la nouvelle image.

Ce workflow utilise des variables d'environnement et des secrets GitHub pour gérer les informations sensibles comme les clés SSH et les identifiants. Il s'assure également que chaque étape est exécutée uniquement si les précédentes ont réussi, garantissant ainsi l'intégrité du processus de déploiement.

Le fichier de workflow complet, avec tous les détails de configuration, sera disponible en annexe pour référence.

Cinquième partie

ÉTUDES DE CAS ET CONCLUSION

Dans cette partie finale, j'explore des études de cas concrètes et des scénarios d'utilisation de notre blog, illustrant les fonctionnalités clés développées. Nous concluons également avec une réflexion sur les résultats obtenus et les perspectives d'avenir.

10 ÉTUDES DE CAS ET SCÉNARIOS D'UTILISATION

10.1 INTRODUCTION

Dans ce chapitre, nous allons explorer quelques scénarios d'utilisation concrets de notre blog, illustrant les fonctionnalités clés que nous avons développées.

10.2 SCÉNARIO 1 : PUBLICATION D'UN NOUVEL ARTICLE

Un utilisateur administrateur souhaite publier un nouvel article sur le blog.

1. L'administrateur se connecte à l'interface d'administration de Strapi.
2. Il navigue vers la section "Articles" et clique sur "Créer un nouvel article".
3. Il remplit les champs nécessaires : titre, contenu, tags associés, etc.
4. Il ajoute des images et formate le contenu selon ses besoins.
5. Une fois satisfait, il clique sur "Publier".
6. L'article est immédiatement visible sur le frontend du blog.

10.3 SCÉNARIO 2 : NAVIGATION ET RECHERCHE PAR UN VISITEUR

Un visiteur cherche des informations sur un sujet spécifique.

1. Le visiteur arrive sur la page d'accueil du blog.
2. Il se rend sur la page Tags et sélectionne une catégorie qui l'intéresse.
3. Les résultats s'affichent, montrant les articles pertinents.
4. Le visiteur clique sur un article qui l'intéresse.
5. Il lit l'article et explore les tags associés pour trouver d'autres contenus similaires.

10.4 SCÉNARIO 3 : MISE À JOUR DU CONTENU

L'administrateur souhaite mettre à jour un article existant.

1. L'administrateur se connecte à Strapi.

2. Il navigue vers la liste des articles et sélectionne celui à modifier.
3. Il effectue les changements nécessaires (correction, ajout d'informations, etc.).
4. Il sauvegarde les modifications.
5. Le contenu mis à jour est automatiquement reflété sur le frontend.

10.5 ANALYSE DES SCÉNARIOS

Ces scénarios démontrent la flexibilité et l'efficacité de notre système de blog. Ils illustrent comment les fonctionnalités clés, telles que la gestion de contenu, la recherche et la mise à jour en temps réel, fonctionnent de manière cohérente pour offrir une expérience utilisateur fluide, tant pour les administrateurs que pour les visiteurs.

11 CONCLUSION

11.1 RÉSUMÉ DES RÉALISATIONS

Au terme de ce projet, j'ai réussi à créer un blog fonctionnel et moderne, répondant aux objectifs initiaux fixés. Les principales réalisations comprennent :

— La mise en place d'une architecture robuste basée sur Strapi pour le backend et Next.js pour le frontend.
— L'implémentation d'un système de gestion de contenu flexible et facile à utiliser.
— Le développement d'une interface utilisateur réactive et esthétique.
— La mise en place d'un système de déploiement continu efficace.
— L'intégration de fonctionnalités de recherche et de navigation par tags.

11.2 DIFFICULTÉS RENCONTRÉES ET SOLUTIONS APPORTÉES

Au cours du développement, nous avons fait face à plusieurs défis :

11.2.1 *Optimisation des Performances*

— **Défi :** Assurer des temps de chargement rapides et une expérience utilisateur fluide.
— **Solution :** Mise en place du rendu côté serveur (SSR) et de la génération statique incrémentale (ISR) avec Next.js.

11.2.2 *Gestion du Déploiement Continu*

— **Défi :** Automatiser le processus de déploiement tout en assurant la qualité du code.
— **Solution :** Configuration d'un pipeline CI/CD avec GitHub Actions et utilisation de Docker pour la conteneurisation.

11.2.3 *Configuration du VPS*

— **Défi :** Mettre en place une infrastructure sécurisée et performante pour héberger le blog.
— **Solution :** Configuration d'un serveur privé virtuel (VPS) incluant l'installation de Docker, la configuration de Nginx comme

serveur web et reverse-proxy, ainsi que la sécurisation du serveur avec des certificats SSL via Let's Encrypt.

11.2.4 *Intégration de Différentes Technologies*

— **Défi :** Assurer une intégration harmonieuse entre les différentes technologies utilisées dans le projet.
— **Solution :** Utilisation de Docker pour containeriser les différents services (Strapi, Next.js, Nginx, etc.), permettant une meilleure gestion et isolation des composants.

11.2.5 *Gestion du Contenu*

11.3 PERSPECTIVES D'AMÉLIORATION ET DE DÉVELOPPEMENT FUTUR

Bien que le projet ait atteint ses objectifs initiaux, plusieurs pistes d'amélioration et de développement futur se dessinent :
— Implémentation d'un système de commentaires pour les articles.
— Intégration d'une fonctionnalité de newsletter pour fidéliser les lecteurs.
— Mise en place d'un CDN pour accélérer le chargement des ressources.
— Utilisation d'un proxy pour masquer l'adresse IP du VPS.
— Développement d'une version mobile native de l'application.
— Implémentation des fonctionnalités de recherche par nom et par date.
— Mise en place d'un système d'analyse de données pour mieux comprendre le comportement des utilisateurs.

11.4 RÉFLEXION FINALE

Ce projet de blog a non seulement permis de créer une plateforme fonctionnelle et moderne, mais a également été une opportunité précieuse d'apprentissage et de mise en pratique de compétences variées en développement web et en gestion de projet. Les défis rencontrés ont stimulé la créativité et la résolution de problèmes, ouvrant la voie à de futures améliorations et innovations.

La réalisation de ce blog représente une étape importante dans mon parcours de développeur, consolidant mes compétences techniques tout en m'offrant une vision plus large des enjeux liés à la conception et au déploiement d'applications web modernes. Les leçons apprises et l'expérience acquise constitueront sans aucun doute une base solide pour mes futurs projets professionnels.

Sixième partie

APPENDIX

A

MAQUETTE DU SITE

A.1 INTRODUCTION

Dans cette annexe, je présente la maquette du site que j'ai réalisé pour le projet de fin d'année.

A.2 ZONING

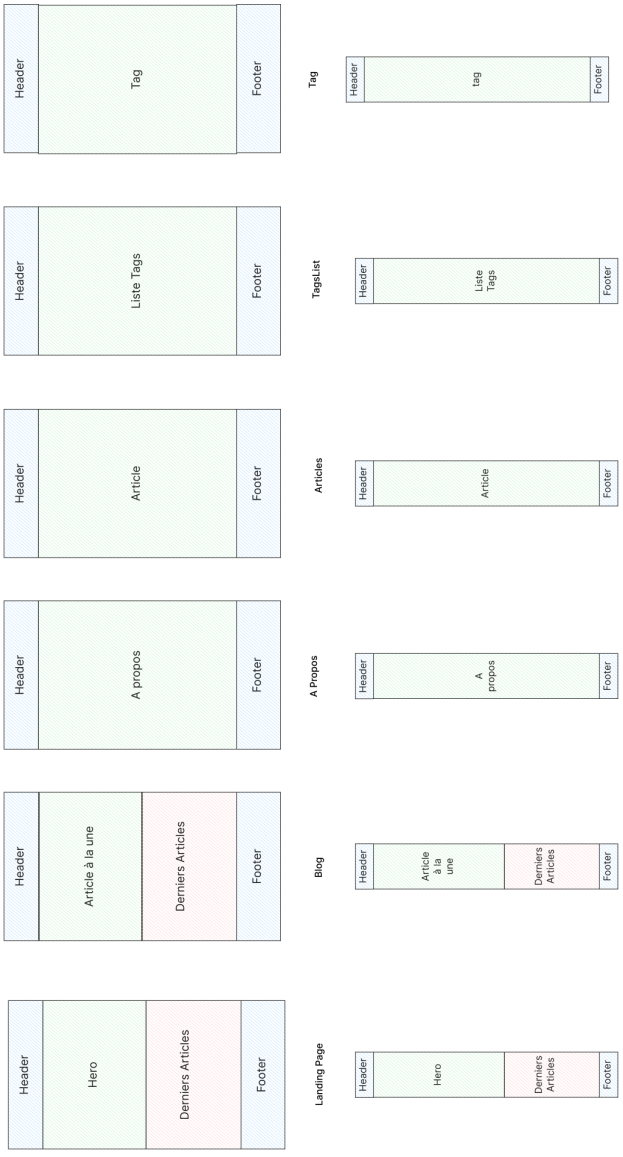


FIGURE 30 – Zoning du site

A.3 WIREFRAME

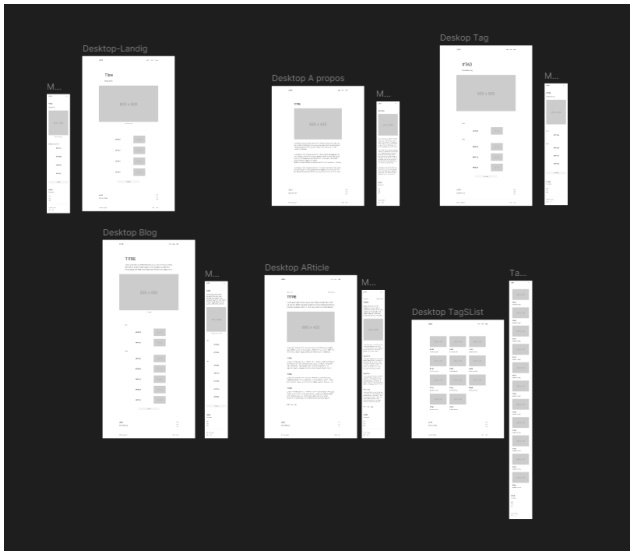


FIGURE 31 – Wireframe du site

A.4 PROTOTYPE

A.4.1 *Accueil*

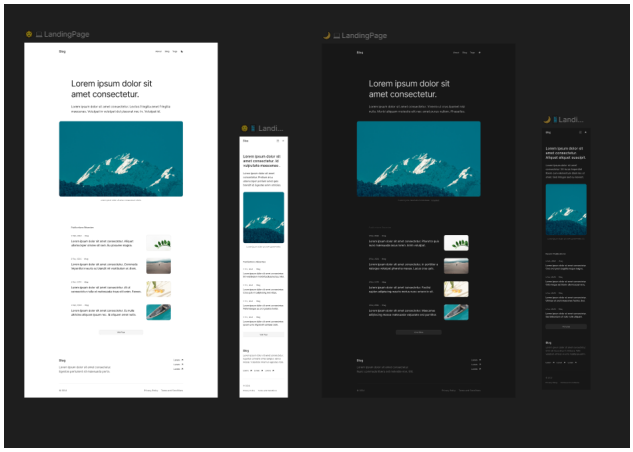


FIGURE 32 – Prototype de la page d'accueil

A.4.2 *Tags*

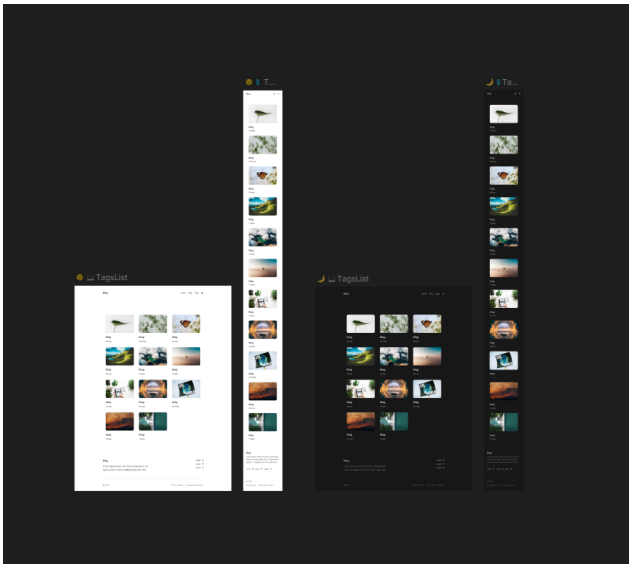


FIGURE 33 – Prototype de la page des tags

A.4.3 *Tag*

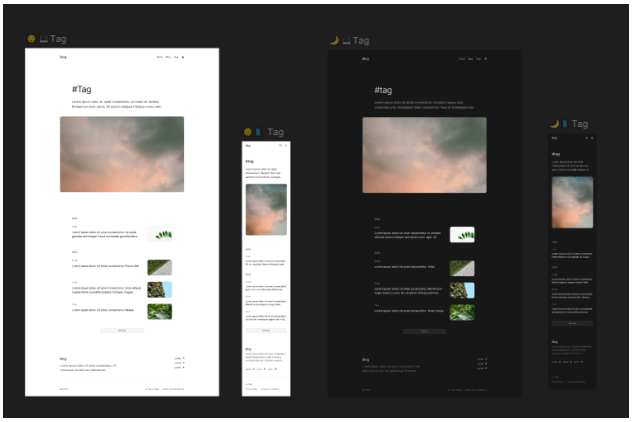


FIGURE 34 – Prototype de la page d'un tag

A.4.4 *Article*

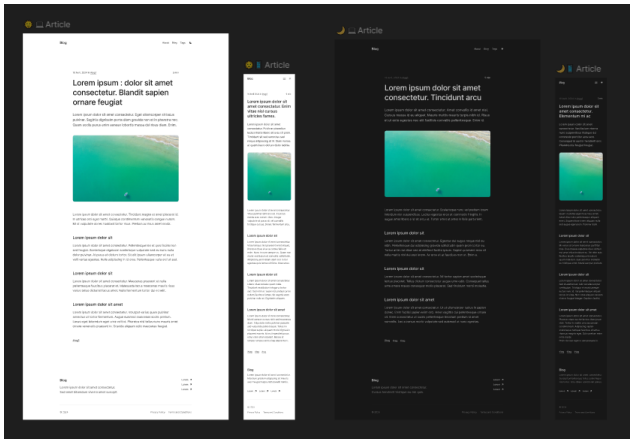


FIGURE 35 – Prototype de la page d'un article

A.4.5 *About*

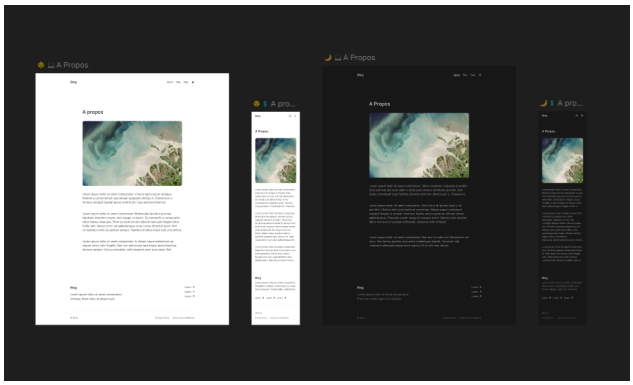


FIGURE 36 – Prototype de la page "À propos"

B

NEXTJS

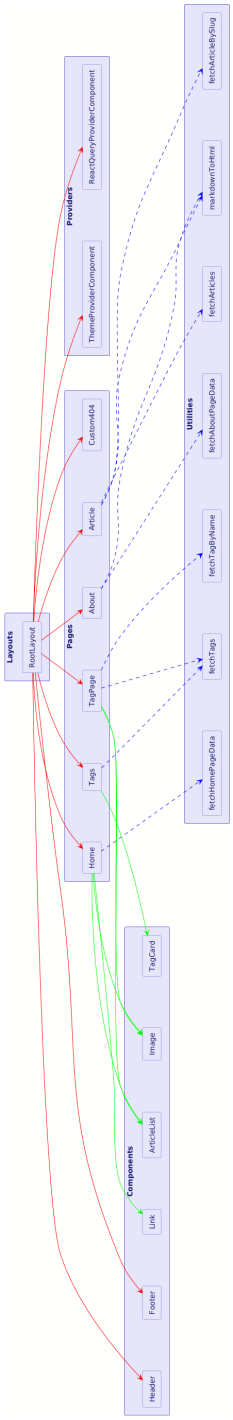


FIGURE 37 – Diagramme des Composants

```ts
import axios, { AxiosError, AxiosInstance } from 'axios';

const createApiClient = (baseURL: string): AxiosInstance => {
  const apiClient = axios.create({
    baseURL,
    headers: {
      "Content-Type": "application/json",
    },
  });

  // Response interceptor for API calls
  apiClient.interceptors.response.use(
    (response) => response, // Return the response unchanged for successful requests
    (error: AxiosError) => {
      // Handle the error
      if (error.response) {
        // The request was made and the server responded with a status code
        // that falls out of the range of 2xx
        console.error("API Error Response:", error.response.data);
        console.error("Status:", error.response.status);
        console.error("Headers:", error.response.headers);
      } else if (error.request) {
        // The request was made but no response was received
        console.error("API Error Request:", error.request);
      } else {
        // Something happened in setting up the request that triggered an Error
        console.error("API Error Message:", error.message);
      }
      console.error("API Error Config:", error.config);

      // You can perform additional actions here, like showing a notification

      // Rethrow the error to be handled by the calling function if needed
      return Promise.reject(error);
    }
  );

  return apiClient;
};

// Create server-side API client
export const apiServer = createApiClient(`${process.env.STRAPI_URL}/api` || '');

// Create client-side API client
export const apiClient = createApiClient('/api');
```

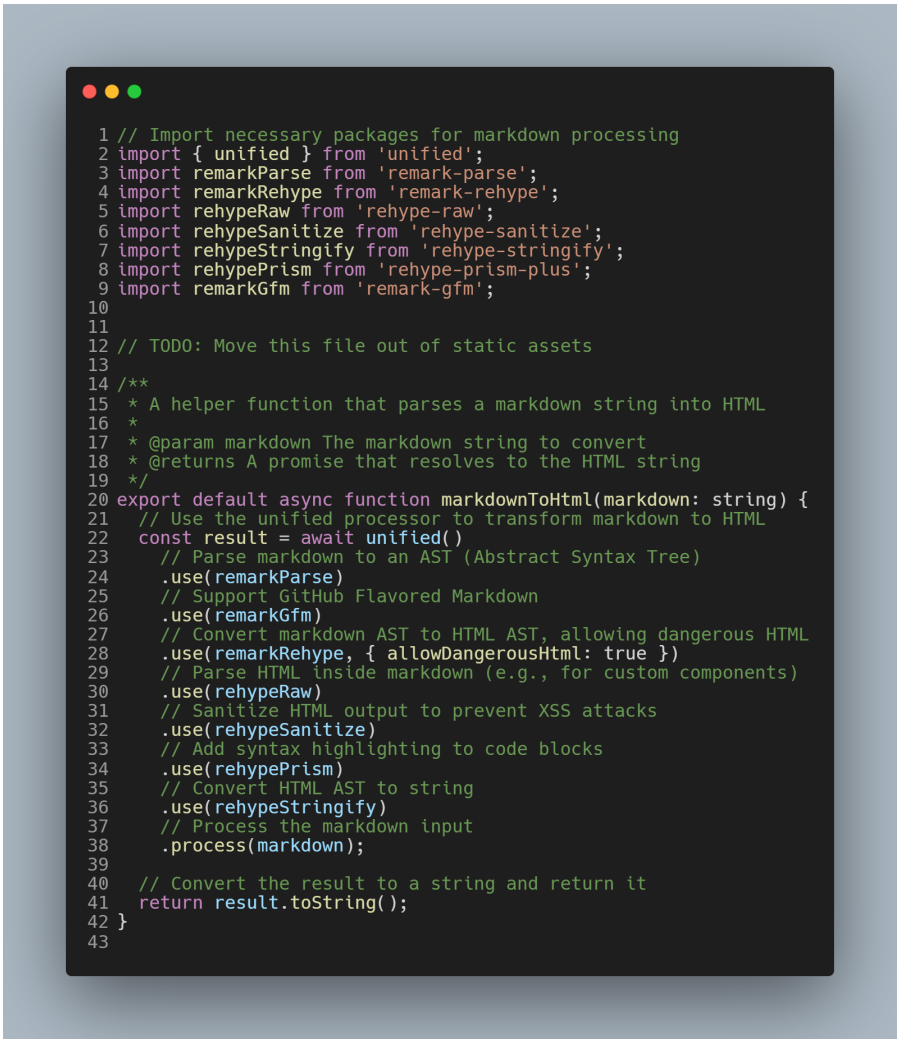
FIGURE 38 – Fichier axiosConfig.ts

```ts
/**
 * This file is used to define API requests that will be used for client-side requests.
 */

import { QueryFunction, QueryFunctionContext } from "@tanstack/react-query";
import { infiniteQueryFormatedData } from "@/interfaces/InfiniteQueryFormatedData";
import { apiClient } from "./axiosConfig";

// Number of articles per page
const PAGE_SIZE = 4;

/**
 * Fetches articles from the API with pagination and populates related data.
 *
 * @param {QueryFunctionContext<[string, string], number>} context - The context object containing page parameters and query key.
 * @returns {Promise<infiniteQueryFormatedData>} The formatted data including articles, current page, and next page if available. This format is directly used by react-query infinite query hook.
 */
export const fetchArticles: QueryFunction<
  infiniteQueryFormatedData,
  [string, string],
  number
> = async ({
  pageParam = 1,
  queryKey,
}: QueryFunctionContext<[string, string], number>) => {
  // TODO: Add axios typed response

  const [_key, queryParams = ""] = queryKey;
  const response = await apiClient.get(
    `/articles?pagination[pageSize]=${PAGE_SIZE}&pagination[page]=${pageParam}&populate[tags][fields][0]=name&populate[coverImage][fields][0]=name&populate[coverImage][fields][1]=alternativeText&populate[coverImage][fields][2]=caption&populate[coverImage][fields][3]=formats&fields[0]=title&fields[1]=description&fields[2]=publishedAt&fields[3]=slug&status=published&${queryParams}`,
  );
  return {
    data: response.data,
    currentPage: pageParam,
    // If the number of pages is greater than the current page, next page is current page + 1, otherwise return null
    nextPage:
      response.data.meta.pagination.pageCount > pageParam
        ? pageParam + 1
        : null,
  };
};

```

FIGURE 39 – Fichier axiosClient.ts

```ts
/**
 *
 * This file is used to define api request that will be used for server side requests
 *
 */


import { Article } from "@/interfaces/Article";
import { BlogHomePage } from "@/interfaces/Blog-home-page";
import { Payload } from "@/interfaces/Payload";
import { Tag } from "@/interfaces/Tag";
import { AxiosResponse } from "axios";
import { apiServer } from "./axiosConfig";
import { About } from "@/interfaces/About";



/**
 * Fetch all the articles
 *
 * @returns A promise that resolves as a payload of Articles
 */
export const fetchArticles = async (): Promise<Payload<Article[]>> => {
  // TODO: Add axios typed response
  const response: AxiosResponse<Payload<Article[]>> = await apiServer.get(
    "/articles?status=published",
  );
  return response.data;
};

/**
 * Fetch the home page data
 *
 * @returns A promise that resolves as a payload containing home page data
 */
export const fetchHomePageData = async (): Promise<Payload<BlogHomePage>> => {
  // TODO: Add axios typed response
  const response = await apiServer.get(
    "/blog-home-page?populate[featuredArticle][populate][0]=coverImage",
  );
  return response.data;
};

/**
 * Fetch the about page data
 *
 * @returns A promise that resolves as a payload containing about page data
 */
export const fetchAboutPageData = async (): Promise<Payload<About>> => {
  const response = await apiServer.get("/about?populate=*");
  return response.data;
};

/**
 * Fetch all the tags
 *
 * @returns A promise that resolves as a payload of Tags
 */
export const fetchTags = async (): Promise<Payload<Tag[]>> => {
  const response = await apiServer.get("/tags?populate=*");
  return response.data;
};

/**
 * Fetch a tag by its name
 *
 * @param name the name of the tag
 * @returns A promise that resolves as a Payload Tags
 */
export const fetchTagByName = async (name: string): Promise<Payload<Tag[]>> => {
  const response = await apiServer.get(
    `/tags?filters[name][$eqi]=${name}&populate=*`,
  );
  return response.data;
};

/**
 * Fetch all the articles associated with a specific tag name
 *
 * @param tagName Name of the tag
 * @params Optional parameters to add to te query
 * @returns
 */
export const fetchArticlesByTagName = async (
  tagName: string,
  params?: string,
): Promise<Payload<Partial<Article>[]>> => {
  const response = await apiServer.get(
    `/articles?filters[tags][name][$in]=${tagName}&${params ?? ""}`,
  );
  return response.data;
};

/**
 * Fetch an article by slug with its Seo data
 *
 * @param slug the slug of the article to retrieve
 * @returns A promise that resolves as a Payload containing a unique article
 */
export const fetchArticleBySlug = async (
  slug: string,
): Promise<Payload<Article[]>> => {
  const response = await apiServer.get(
    `/articles?filters[slug][$eqi]=${slug}&populate[seo][populate][1]=metaImage&populate[seo][populate][2]=metaTwitterImage&populate=tags`,
  );
  return response.data;
};

```

FIGURE 40 – Fichier axiosServer.ts

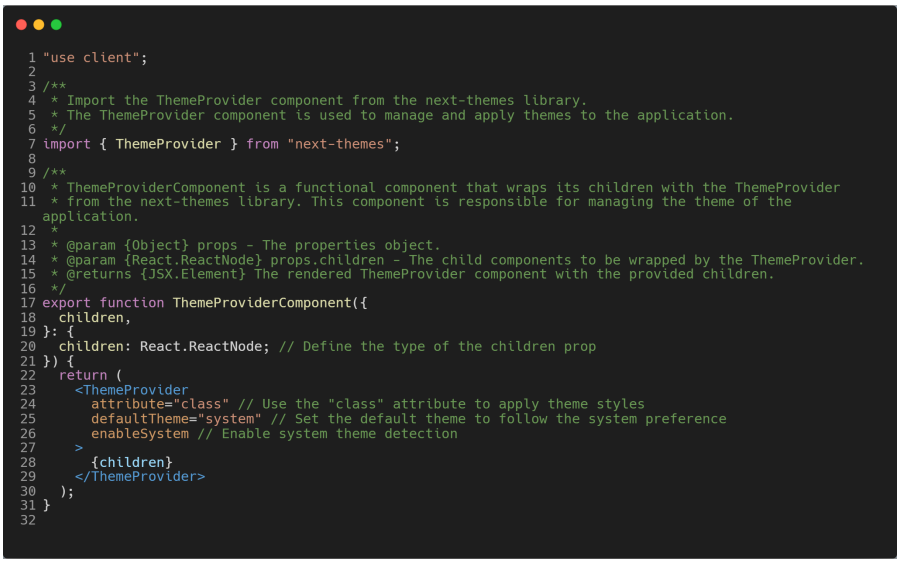


```ts
// Import necessary packages for markdown processing
import { unified } from 'unified';
import remarkParse from 'remark-parse';
import remarkRehype from 'remark-rehype';
import rehypeRaw from 'rehype-raw';
import rehypeSanitize from 'rehype-sanitize';
import rehypeStringify from 'rehype-stringify';
import rehypePrism from 'rehype-prism-plus';
import remarkGfm from 'remark-gfm';


// TODO: Move this file out of static assets

/**
 * A helper function that parses a markdown string into HTML
 *
 * @param markdown The markdown string to convert
 * @returns A promise that resolves to the HTML string
 */
export default async function markdownToHtml(markdown: string) {
  // Use the unified processor to transform markdown to HTML
  const result = await unified()
    // Parse markdown to an AST (Abstract Syntax Tree)
    .use(remarkParse)
    // Support GitHub Flavored Markdown
    .use(remarkGfm)
    // Convert markdown AST to HTML AST, allowing dangerous HTML
    .use(remarkRehype, { allowDangerousHtml: true })
    // Parse HTML inside markdown (e.g., for custom components)
    .use(rehypeRaw)
    // Sanitize HTML output to prevent XSS attacks
    .use(rehypeSanitize)
    // Add syntax highlighting to code blocks
    .use(rehypePrism)
    // Convert HTML AST to string
    .use(rehypeStringify)
    // Process the markdown input
    .process(markdown);

  // Convert the result to a string and return it
  return result.toString();
}

```

FIGURE 41 – Fichier markdownToHtml.ts

```tsx
"use client";

import React from "react";
import { QueryClientProvider, QueryClient } from "@tanstack/react-query";
import { ReactQueryDevtools } from "@tanstack/react-query-devtools";

/**
 * ReactQueryProviderComponent is a provider component that sets up the React Query client
 * and provides it to the rest of the application. It also includes the React Query Devtools
 * for debugging purposes.
 *
 * @param {React.PropsWithChildren} props - The properties object containing the children
 * components.
 * @returns {JSX.Element} The rendered QueryClientProvider component with React Query Devtools.
 */
function ReactQueryProviderComponent({ children }: React.PropsWithChildren) {
  // Initialize the QueryClient instance using React's useState hook
  const [client] = React.useState(new QueryClient());

  return (
    // Provide the QueryClient instance to the rest of the application
    <QueryClientProvider client={client}>
      {children}
      {/* Include the React Query Devtools for debugging */}
      <ReactQueryDevtools initialIsOpen={false} />
    </QueryClientProvider>
  );
}

export default ReactQueryProviderComponent;

```

FIGURE 42 – Fichier ReactQueryProviderComponent.tsx



```tsx
"use client";

/**
 * Import the ThemeProvider component from the next-themes library.
 * The ThemeProvider component is used to manage and apply themes to the application.
 */
import { ThemeProvider } from "next-themes";

/**
 * ThemeProviderComponent is a functional component that wraps its children with the ThemeProvider
 * from the next-themes library. This component is responsible for managing the theme of the application.
 *
 * @param {Object} props - The properties object.
 * @param {React.ReactNode} props.children - The child components to be wrapped by the ThemeProvider.
 * @returns {JSX.Element} The rendered ThemeProvider component with the provided children.
 */
export function ThemeProviderComponent({
  children,
}: {
  children: React.ReactNode; // Define the type of the children prop
}) {
  return (
    <ThemeProvider
      attribute="class" // Use the "class" attribute to apply theme styles
      defaultTheme="system" // Set the default theme to follow the system preference
      enableSystem // Enable system theme detection
    >
      {children}
    </ThemeProvider>
  );
}

```

FIGURE 43 – Fichier ThemeProviderComponent.tsx

```ts
import { Media } from "./Media";
import { Media_Plain } from "./Media";
import { Article } from "./Article";
import { Article_Plain } from "./Article";
import { Seo, Seo_NoRelations, Seo_Plain } from "./Seo";

export interface Tag {
  id: number;
  attributes: {
    createdAt: Date | string;
    updatedAt: Date | string;
    publishedAt?: Date | string;
    name: string;
    cover: { data: Media };
    articles?: { data: Article[] };
    description?: string;
    seo?: Seo;
    locale: string;
    localizations?: { data: Tag[] };
  };
}
export interface Tag_Plain {
  id: number;
  createdAt: Date | string;
  updatedAt: Date | string;
  publishedAt?: Date | string;
  name: string;
  cover: Media_Plain;
  articles?: Article_Plain[];
  description?: string;
  seo?: Seo_Plain
  locale: string;
  localizations?: Tag_Plain[];
}

export interface Tag_NoRelations {
  id: number;
  createdAt: Date | string;
  updatedAt: Date | string;
  publishedAt?: Date | string;
  name: string;
  cover: number;
  articles?: number[];
  description?: string;
  seo?: Seo_NoRelations
  locale: string;
  localizations?: Tag[];
}

```

FIGURE 44 – Fichier Tag.ts

```
export interface Payload<T>
{     data: T;
      meta: {
        pagination?: {
          page: number;
          pageSize: number;
          pageCount: number;
          total: number;
        }
      };
    }
```

FIGURE 45 – Fichier Payload.ts

```tsx
import { fetchAboutPageData } from "@/libs/axiosServer";
import markdownToHtml from "@/libs/markdownToHtml";
import { Metadata } from 'next';

export async function generateMetadata(): Promise<Metadata> {
  const fetchedData = await fetchAboutPageData();
  const seo = fetchedData.data.attributes.seo;

  return {
    title: seo?.metaTitle || 'À propos | Aniss.dev Blog',
    description: seo?.metaDescription || 'En savoir plus sur Aniss et son blog',
    openGraph: {
      title: seo?.metaTitle,
      description: seo?.metaDescription,
      images: seo?.metaImage?.data?.attributes.url ? [seo.metaImage.data.attributes.url] : [],
    },
  };
}

const About = async () => {
  try {
    const aboutPageDataPayload = await fetchAboutPageData();

    // Check if the about page data exists
    if (!aboutPageDataPayload.data) {
      throw new Error('About page data not found');
    }

    const aboutData = aboutPageDataPayload.data;

    // Parse the markdown content to HTML
    const content = await markdownToHtml(aboutData.attributes.content || '');

    return (
      <main className="container flex flex-col gap-8 lg:max-w-screen-lg">
        <div className="lg:px-10 xl:px-20">
          <h1 className="mb-6 text-2xl font-medium leading-normal text-neutral-900 dark:text-neutral-
300 sm:text-4xl sm:leading-10 lg:mb-8 lg:text-5xl xl:text-6xl">
            À propos
          </h1>
          <div
            className="prose prose-lg dark:prose-invert "
            dangerouslySetInnerHTML={{ __html: content || '' }}
          />
        </div>
      </main>
    );

  } catch (error) {
    console.error("Error fetching about page data:", error);
    return (
      <main className="container flex flex-col gap-8 lg:max-w-screen-lg">
        <div className="lg:px-10 xl:px-20">
          <h1 className="mb-6 text-2xl font-medium leading-normal text-neutral-900 dark:text-neutral-
300 sm:text-4xl sm:leading-10 lg:mb-8 lg:text-5xl xl:text-6xl">
            Error
          </h1>
          <p>Une erreur s'est produite lors de la récupération des données de la page À propos.
Veuillez réessayer plus tard.</p>
        </div>
      </main>
    );
  }

};

export default About;

```

FIGURE 46 – Fichier About.tsx

C

LES TESTS

C.1 INTRODUCTION

Cette partie vise à présenter certains tests que j'ai réalisés pour valider la qualité et la fiabilité de mon application.

C.2 LES TESTS UNITAIRES AVEC LE COMPOSANT ARTICLECARD

Plan de Test 1 : Vérification du rendu du titre de l'article

Objectif : S'assurer que le titre de l'article est affiché correctement.

Étapes :

1. Rendre le composant `ArticleCard` avec l'article simulé (`mockArticle`).
2. Vérifier que le texte "Article test" est présent dans le document.

Résultat attendu : Le titre "Article test" doit être visible dans le document.

Plan de Test 2 : Vérification de l'affichage des tags de l'article

Objectif : S'assurer que tous les tags de l'article sont affichés.

Étapes :

1. Rendre le composant `ArticleCard` avec l'article simulé.
2. Pour chaque tag dans `mockArticle.attributes.tags.data`, vérifier que le texte correspondant est présent dans le document.

Résultat attendu : Chaque tag doit être affiché sous la forme #nomDuTag.

Plan de Test 3 : Vérification de l'affichage de la date de l'article

Objectif : S'assurer que la date de l'article est affichée correctement.

Étapes :

1. Rendre le composant `ArticleCard` avec l'article simulé.
2. Utiliser une fonction personnalisée pour trouver l'élément de date qui est un tag `h3` et qui contient le jour "10" et le mois "sep".
3. Vérifier que cet élément est présent dans le document.

Résultat attendu : L'élément de date doit être visible et contenir les informations correctes.

Plan de Test 4 : Vérification du rendu du résumé de l'article

Objectif : S'assurer que le résumé de l'article est affiché.

Étapes :

1. Rendre le composant `ArticleCard` avec l'article simulé.
2. Vérifier que le texte "This is a test article summary" est présent dans le document.

Résultat attendu : Le résumé doit être visible dans le document.

Plan de Test 5 : Vérification de l'affichage de l'image de l'article

Objectif : S'assurer que l'image de l'article est affichée avec les attributs corrects.

Étapes :

1. Rendre le composant `ArticleCard` avec l'article simulé.
2. Trouver l'élément image dans le document.
3. Vérifier que l'image a les attributs `src` et `alt` corrects.

Résultat attendu : L'image doit avoir `src="https://example.com/image.jpg"` et `alt="CoverImage"`.

Plan de Test 6 : Vérification de la correspondance avec le snapshot

Objectif : S'assurer que le composant `ArticleCard` correspond au snapshot enregistré.

Étapes :

1. Rendre le composant `ArticleCard` avec l'article simulé.
2. Comparer le fragment rendu avec le snapshot existant.

Résultat attendu : Le fragment rendu doit correspondre au snapshot enregistré.

C.2.0.1 *Le code*

```tsx
// Import necessary dependencies
import React from "react";
import { render, screen } from "@testing-library/react";
import ArticleCard from "../src/components/ui/article-card/ArticleCard";
import { mockArticle } from "../__mocks__/mockArticle";


describe("ArticleCard unit tests", () => {
  // Test case: Verify that the article title is rendered correctly
  it("renders the article title", () => {
    render(<ArticleCard article={mockArticle} />);
    expect(screen.getByText("Article test")).toBeInTheDocument();
  });

  // Test case: Check if all article tags are displayed
  it("displays the article tags", () => {
    // Render the ArticleCard component with the mock article
    render(<ArticleCard article={mockArticle} />);

    // Iterate through each tag in the mockArticle's tags data
    mockArticle.attributes.tags.data.forEach((tag) => {
      // For each tag, check if its name is rendered in the document
      // The tag name should be prefixed with a '#' symbol
      expect(screen.getByText(`#${tag.attributes.name}`)).toBeInTheDocument();
    });
  });

  // Test case: Ensure the article date is shown correctly
  it("shows the article date", () => {
    // Render the ArticleCard component with the mock article
    render(<ArticleCard article={mockArticle} />);

    // Use a custom function to find the date element
    const dateElement = screen.getByText((content, element) => {
      // Ensure the element is an h3 tag
      if (element?.tagName.toLowerCase() !== "h3") return false;

      // Check if the content includes the day (10)
      const hasDay = content.includes("10");

      // Check if the element's text content includes the month (sep)
      const hasMonth = element.textContent?.toLowerCase().includes("sep");

      // Return true if both day and month are present or false if not so we are sure to return a
booleareturn (hasDay && hasMonth) || false;
    });

    // Assert that the date element is in the document
    expect(dateElement).toBeInTheDocument();
  });

  // Test case: Verify that the article summary is rendered
  it("renders the article summary", () => {
    // Render the ArticleCard component with the mock article
    render(<ArticleCard article={mockArticle} />);

    // Check if the article summary is rendered in the document
    expect(
      screen.getByText("This is a test article summary"),
    ).toBeInTheDocument();
  });

  // Test case: Check if the article image is displayed with correct attributes
  it("displays the article image", () => {
    // Render the ArticleCard component with the mock article
    render(<ArticleCard article={mockArticle} />);

    // Check if the article image is rendered in the document
    const image = screen.getByRole("img");
    expect(image).toHaveAttribute("src", "https://example.com/image.jpg");
    expect(image).toHaveAttribute("alt", "CoverImage");
  })

  // Test case: Ensure the component matches the snapshot
  it("matches snapshot", () => {
    // Render the ArticleCard component with the mock article
    const { asFragment } = render(<ArticleCard article={mockArticle} />);

    // Check if the snapshot matches the rendered component
    expect(asFragment()).toMatchSnapshot();
  });

});

```

FIGURE 47 – Les tests unitaires de ArticleCard

C.3 LES TESTS D'INTÉGRATION

C.3.1 *L'intégration de TagCard avec TagPage*

C.3.1.1 *Les plans de tests*

Plan de Test 1 : Vérification du rendu de tous les tags

Objectif : Vérifier que tous les tags sont correctement rendus sur la page.

Étapes :

1. Simuler l'appel à `fetchTags` pour retourner une liste de tags fictifs (`mockSeveralTags`).
2. Rendre le composant `Tags`.
3. Utiliser `findAllByRole` pour récupérer tous les éléments `article` sur la page.

Résultat attendu : Le nombre d'éléments `article` rendus doit correspondre au nombre de tags dans `mockSeveralTags`.

Plan de Test 2 : Vérification du rendu des composants TagCard avec les données correctes

Objectif : Vérifier que chaque composant `TagCard` est rendu avec les données correctes.

Étapes :

1. Simuler l'appel à `fetchTags` pour retourner `mockSeveralTags`.
2. Rendre le composant `Tags`.
3. Pour chaque tag dans `mockSeveralTags` :
   a) Vérifier que le nom du tag est affiché.
   b) Vérifier que le nombre d'articles est affiché.
   c) Vérifier que l'image du tag est affichée avec les attributs corrects.

Résultat attendu : Chaque `TagCard` doit afficher le nom, le nombre d'articles, et l'image avec les attributs corrects.

Plan de Test 3 : Vérification des liens corrects dans les composants TagCard

Objectif : Vérifier que chaque `TagCard` contient un lien avec l'attribut `href` correct.

Étapes :

1. Simuler l'appel à `fetchTags` pour retourner `mockSeveralTags`.
2. Rendre le composant `Tags`.

3. Pour chaque tag dans `mockSeveralTags` :
   a) Trouver le `TagCard` correspondant.
   b) Vérifier que le lien dans le `TagCard` a l'attribut `href` correct.

Résultat attendu : Chaque `TagCard` doit contenir un lien avec l'attribut `href` pointant vers `/tags/{nom_du_tag}`.

Plan de Test 4 : Vérification du nombre correct de composants TagCard rendus

Objectif : Vérifier que le nombre de composants `TagCard` rendus correspond au nombre de tags.

Étapes :

1. Simuler l'appel à `fetchTags` pour retourner `mockSeveralTags`.
2. Rendre le composant `Tags`.
3. Utiliser `getAllByRole` pour récupérer tous les éléments `article`.

Résultat attendu : Le nombre d'éléments `article` doit correspondre au nombre de tags dans `mockSeveralTags`.

Plan de Test 5 : Vérification de la correspondance avec le snapshot

Objectif : Vérifier que le rendu du composant `Tags` correspond au snapshot enregistré.

Étapes :

1. Simuler l'appel à `fetchTags` pour retourner `mockSeveralTags`.
2. Rendre le composant `Tags`.
3. Comparer le fragment rendu avec le snapshot existant.

Résultat attendu : Le fragment rendu doit correspondre exactement au snapshot enregistré.

C.3.1.2 *Le code*

```tsx
import { render, screen } from "@testing-library/react";
import Tags from "../src/app/tags/page";
import { fetchTags } from "@/libs/axiosServer";
import { mockSeveralTags } from "../__mocks__/mockSeveralTags";

// Mock the fetchTags function
jest.mock("@/libs/axiosServer", () => ({
  fetchTags: jest.fn(),
}));

// Mock the fetchTags function to return mockSeveralTags
(fetchTags as jest.Mock).mockResolvedValue({ data: mockSeveralTags });

/**
 * Integration tests for the Tags Page component
 */
describe("Tags Page and TagCard Integration Tests", () => {
  /**
   * Test case: Verify that all tags are rendered
   */
  it("renders all tags", async () => {
    // Arrange: Set up the expected tag count
    const expectedTagCount = mockSeveralTags.length;

    // Act: Render the Tags component and find all article elements
    const { findAllByRole } = render(await Tags({}));
    const tagCards = await findAllByRole("article");

    // Assert: Check if the number of rendered tag cards matches the expected
counexpect(tagCards).toHaveLength(expectedTagCount);
  });

  /**
   * Test case: Verify that TagCard components are rendered with correct data
   */
  it("renders TagCard components with correct data", async () => {
    // Arrange: Set up the expected tags data
    const expectedTags = mockSeveralTags;

    // Act: Render the Tags component
    render(await Tags({}));

    // Assert: Check if each tag's data is correctly rendered
    for (const tag of expectedTags) {
      // Check if tag name is rendered
      expect(screen.getByText(`#${tag.attributes.name}`)).toBeInTheDocument();
      // Check if article count is rendered
      expect(
        screen.getByText(`${tag.attributes.articles?.data.length} Articles`),
      ).toBeInTheDocument();

      // Check if tag image is rendered with correct attributes
      const image = screen.getByAltText(
        tag.attributes.cover.data.attributes.alternativeText,
      );
      expect(image).toBeInTheDocument();
      expect(image).toHaveAttribute(
        "src",
        expect.stringContaining(
          tag.attributes.cover.data.attributes.formats.thumbnail.url,
        ),
      );
    }
  });

  /**
   * Test case: Verify that TagCard components have correct links
   */
  it("renders TagCard components with correct links", async () => {
    // Arrange: Set up the expected tags data
    const expectedTags = mockSeveralTags;

    // Act: Render the Tags component and get all article elements
    render(await Tags({}));
    const tagCards = screen.getAllByRole("article");

    // Assert: Check if each tag card has the correct link
    for (const tag of expectedTags) {
      // Find the tag card for the current tag
      const tagCard = tagCards.find((card) =>
        card.textContent?.includes(`#${tag.attributes.name}`),
      );
      expect(tagCard).toBeTruthy();

      if (tagCard) {
        // Check if the link in the tag card has the correct href attribute
        const link = tagCard.querySelector("a");
        expect(link).toHaveAttribute(
          "href",
          `/tags/${tag.attributes.name.toLowerCase()}`,
        );
      }
    }
  });

  /**
   * Test case: Verify the correct number of TagCard components are rendered
   */
  it("renders the correct number of TagCard components", async () => {
    // Arrange: Set up the expected tag count
    const expectedTagCount = mockSeveralTags.length;

    // Act: Render the Tags component and get all article elements
    render(await Tags({}));
    const tagCards = screen.getAllByRole("article");

    // Assert: Check if the number of rendered tag cards matches the expected
counexpect(tagCards).toHaveLength(expectedTagCount);
  });

  /**
   * Test case: Verify that the Tags component matches the snapshot
   */
  it("matches snapshot", async () => {
    // Arrange & Act: Render the Tags component
    const { asFragment } = render(await Tags({}));

    // Assert: Check if the rendered component matches the snapshot
    expect(asFragment()).toMatchSnapshot();
  });
});

```

FIGURE 48 – Les tests d'integration de TagCard avec TagPage

C.4 LES TESTS DE CHARGE

C.4.1 *Test de Charge de Base*

Objectif : Vérifier si le blog peut gérer un nombre modéré d'utilisateurs simultanés.

Configuration :

— Nombre d'utilisateurs virtuels (VU) : 50
— Durée : 5 minutes

Scénario :

— Chaque utilisateur virtuel (VU) accède à la page d'accueil du blog.
— Les utilisateurs attendent entre 1 et 3 secondes avant de faire une autre requête.

Critères de Réussite :

— Temps de réponse moyen inférieur à 500 ms.
— Moins de 1 % des requêtes échouent.

C.4.2 *Test d'Endurance*

Objectif : Tester la stabilité du blog sur une longue période avec une charge constante.

Configuration :

— Nombre d'utilisateurs virtuels (VU) : 100
— Durée : 30 minutes

Scénario :

— Les utilisateurs accèdent à l'accueil et naviguent entre les différentes catégories du blog.
— Lecture de 3 à 5 articles par utilisateur.
— Pause de 3 à 8 secondes entre chaque action.

Critères de Réussite :

— Pas de dégradation significative des performances sur la durée du test.
— Latence moyenne inférieure à 800 ms.

C.4.3 *Test de Montée en Charge avec Pic*

C.4.3.1 *Le plan de test*

Objectif : Évaluer la capacité du blog à gérer un pic soudain de trafic et à se rétablir.

Configuration :

— Scénario de montée en charge avec pic :
  — 0 à 200 VU en 30 secondes
  — 200 à 2000 VU en 2 minutes
  — Maintien à 2000 VU pendant 1 minute

— Descente de 2000 à 500 VU en 1 minute
— Descente de 500 à 0 VU en 30 secondes
— Durée totale : 5 minutes

Scénario :

— Les utilisateurs visitent la page d'accueil.
— Ils consultent aléatoirement 3 articles.
— Ils naviguent vers une section aléatoire du blog.
— Pause de 1 à 3 secondes entre chaque action.

Critères de Réussite :

— Taux de succès des requêtes supérieur à 90%.
— Temps de réponse moyen inférieur à 2 secondes.
— Le système doit se rétablir rapidement après le pic de charge.

C.4.3.2 *Le resultat*

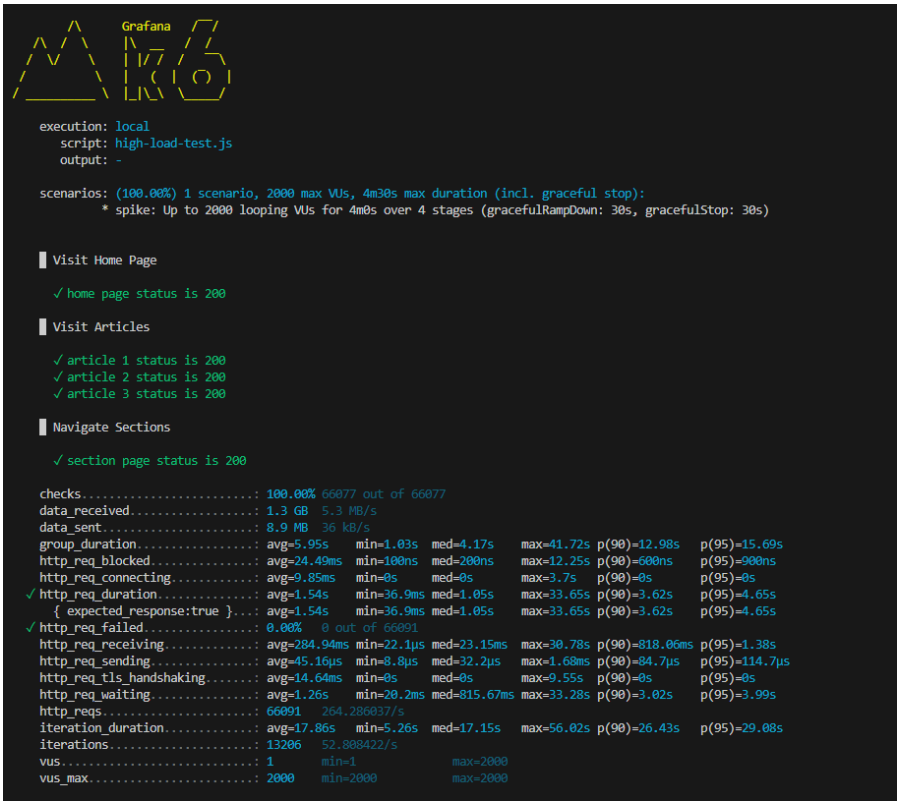


FIGURE 49 – Le resultats k6

FIGURE 50 – Les statistiques des conteneurs

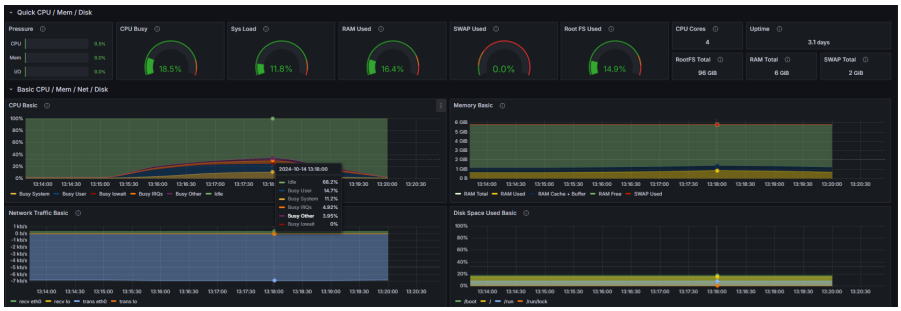


FIGURE 51 – Les statistiques de l'hôte

D

FICHIERS DE CONFIGURATION

D.1 DOCKER FILE NEXT.JS

```
# Use Node.js as the base image
FROM node:18-alpine AS base

# Install dependencies only when needed
FROM base AS deps
# Check https://github.com/nodejs/docker-node/tree/b4117f9333da4138b03a546ec926ef50a31506c3#nodealpine
to understand why libc6-compat might be needed.
RUN apk add --no-cache libc6-compat
WORKDIR /app

# Copy package.json and yarn.lock
COPY package.json yarn.lock ./

# Install dependencies
RUN yarn install --frozen-lockfile

# Rebuild the source code only when needed
FROM base AS builder

ARG STRAPI_URL=https://strapi.aniss.dev
ENV STRAPI_URL=${STRAPI_URL}

WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .

# Set environment variables
ENV NEXT_TELEMETRY_DISABLED 1

# Build the application
RUN yarn build

# Production image, copy all the files and run next
FROM base AS runner
WORKDIR /app

ENV PORT 3000
ENV NODE_ENV production
ENV NEXT_TELEMETRY_DISABLED 1

RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

COPY --from=builder /app/public ./public

# Set the correct permission for prerender cache
RUN mkdir .nextRUN chown nextjs:nodejs .next

# Automatically leverage output traces to reduce image size
# https://nextjs.org/docs/advanced-features/output-file-tracing
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static

USER nextjs

EXPOSE 3000


# server.js is created by next build from the standalone output
# https://nextjs.org/docs/pages/api-reference/next-config-js/output
CMD ["node", "server.js"]

```

FIGURE 52 – Docker File Next.js

D.2 DOCKER FILE STRAPI

```
# Creating multi-stage build for production
FROM node:18-alpine as build
RUN apk update && apk add --no-cache build-base gcc autoconf automake zlib-dev libpng-dev vips-dev > /dev/null 2>&1
ARG NODE_ENV=production
ENV NODE_ENV=${NODE_ENV}

WORKDIR /opt/
COPY package.json yarn.lock ./
RUN yarn config set network-timeout 600000 -g && yarn install --production
ENV PATH /opt/node_modules/.bin:$PATH
WORKDIR /opt/app
COPY . .
RUN yarn build

# Creating final production image
FROM node:16-alpine
RUN apk add --no-cache vips-dev
ARG NODE_ENV=production
ENV NODE_ENV=${NODE_ENV}
WORKDIR /opt/
COPY --from=build /opt/node_modules ./node_modules
WORKDIR /opt/app
COPY --from=build /opt/app ./
ENV PATH /opt/node_modules/.bin:$PATH

RUN chown -R node:node /opt/app
USER node
EXPOSE 1337
CMD ["yarn", "start"]
```

FIGURE 53 – Docker File Strapi

D.3 DOCKER COMPOSE

```yaml
services:
 # Strapi application service
  strapi:
    container_name: strapi
    image: ghcr.io/mowee59/strapi:latest # Use the latest version of the image from the github registry
    pull_policy: always # Always pull the latest image
    restart: unless-stopped # Restart the container unless stopped
    env_file: .env # Use the .env file
    environment:
      # Database and authentication configuration, thoses variables are needed by strapi
      DATABASE_CLIENT: ${DATABASE_CLIENT}
      DATABASE_HOST: strapiDB
      DATABASE_NAME: ${DATABASE_NAME}
      DATABASE_USERNAME: ${DATABASE_USERNAME}
      DATABASE_PORT: ${DATABASE_PORT}
      JWT_SECRET: ${JWT_SECRET}
      ADMIN_JWT_SECRET: ${ADMIN_JWT_SECRET}
      DATABASE_PASSWORD: ${DATABASE_PASSWORD}
      NODE_ENV: ${NODE_ENV}
    volumes:
      # Mount configuration and uploads
      - ./strapi/config:/opt/app/config # Mount the config folder
      #- ./strapi/src:/opt/app/src Uncomment to access the source code
      #- ./strapi/package.json:/opt/package.json Uncomment to access the package.json
      #- ./strapi/yarn.lock:/opt/yarn.lock Uncomment to access the yarn.lock
      #- ./strapi/.env:/opt/app/.env # Mount the .env file
      - ./strapi/public/uploads:/opt/app/public/uploads # Mount the uploads folder
    networks:
      - app-network # Connect to the app network
      - backend-network # Connect to the backend network
    depends_on:
      - strapiDB # Ensure the database is running before starting the application

  # PostgreSQL database service for Strapi
  strapiDB:
    container_name: strapiDB
    platform: linux/amd64 # Specified for compatibility with Apple M1 chips
    restart: unless-stopped # Restart the container unless stopped
    env_file: .env # Use the .env file
    image: postgres:14.5-alpine
    environment:
      # Database credentials
      POSTGRES_USER: ${DATABASE_USERNAME}
      POSTGRES_PASSWORD: ${DATABASE_PASSWORD}
      POSTGRES_DB: ${DATABASE_NAME}
    volumes:
      - strapi-data:/var/lib/postgresql/data/ # Persistent storage for database, using Docker Volume
    networks:
      - backend-network # Connect to the backend network
      - database-network # Connect to the database network

  # Adminer for database management
  strapiAdminer:
    container_name: strapiAdminer
    image: adminer
    restart: unless-stopped
    environment:
      - ADMINER_DEFAULT_SERVER=strapiDB
    networks:
      - database-network # Connect to the database network
    depends_on:
      - strapiDB # Ensure the database is running before starting the adminer

  # Next.js application service
  blogFront:
    container_name: blogFront
    image: ghcr.io/mowee59/blog-frontend:latest
    pull_policy: always # Always pull the latest image
    restart: unless-stopped # Restart the container unless stopped
    env_file: ./blogFront/.env # Use the .env file
    environment:
      - NODE_ENV=${NODE_ENV} # Set the NODE_ENV environment variable
      - STRAPI_URL=${STRAPI_URL} # Set the STRAPI_URL environment variable
    networks:
      - app-network # Connect to the app network
    depends_on:
      - strapi # Ensure the strapi is running before starting the next.js application

  # Nginx reverse proxy
  nginx:
    image: nginx:alpine
    container_name: nginx-proxy
    restart: unless-stopped # Restart the container unless stopped
    ports:
      - "80:80" # Expose port 80 to the host
      - "443:443" # Expose port 443 to the host
    volumes:
      # Mount various configuration and certificate files
      - ./nginx/conf:/etc/nginx/conf.d # Mount the configuration folder
      - ./nginx/certs:/etc/nginx/certs # Mount the certificates folder
      - ./nginx/html:/usr/share/nginx/html # Mount the html folder
      - ./nginx/dhparam:/etc/nginx/dhparam # Mount the dhparam folder
      - ./nginx/vhost:/etc/nginx/vhost.d # Mount the vhost folder
      - ./certbot/www:/var/www/certbot # Mount the webroot folder
      - /var/run/docker.sock:/tmp/docker.sock:ro # Mount the docker socket
    networks:
      - app-network # Connect to the app network
      - cert-network # Connect to the cert network
      - database-network # Connect to adminer network
```

FIGURE 54 – Docker Compose partie 1

```yaml
# Certbot for SSL certificate management
  certbot:
    image: certbot/certbot
    container_name: certbot
    volumes:
      - ./nginx/certs:/etc/letsencrypt:rw # Mount the certificates folder
      - ./certbot/www:/var/www/certbot:rw # Mount the webroot folder
    command: certonly --webroot -w /var/www/certbot --keep-until-expiring --email contact@aniss.dev -d aniss.dev -d www.aniss.dev -d strapi.aniss.dev -d adminer.aniss.dev --agree-tos # Command to generate the certificates
    depends_on:
      - nginx # Ensure the nginx is running before starting the certbot
    networks:
      - cert-network # Connect to the app network


## monitoring
  # Prometheus: Time series database for storing metrics
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus  # Mount local prometheus config
      - prometheus_data:/prometheus   # Persistent storage for prometheus data
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'  # Specify config file location
      - '--storage.tsdb.path=/prometheus'               # Specify data storage location
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--web.enable-lifecycle'                        # Enable runtime reloading of config
    networks:
      - monitoring-network
    restart: unless-stopped

  # Node Exporter: Collects system metrics from the host machine
  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    volumes:
      - /proc:/host/proc:ro           # Mount host /proc (read-only)
      - /sys:/host/sys:ro             # Mount host /sys (read-only)
      - /:/rootfs:ro                  # Mount host root filesystem (read-only)
    command:
      - '--path.procfs=/host/proc'    # Specify path for proc metrics
      - '--path.rootfs=/rootfs'       # Specify path for root filesystem metrics
      - '--path.sysfs=/host/sys'      # Specify path for sys metrics
      - '--collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($$|/)'  # Exclude certain mount points
    networks:
      - monitoring-network
    restart: unless-stopped

  # cAdvisor: Collects container metrics
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    volumes:
      - /:/rootfs:ro                  # Mount host root filesystem (read-only)
      - /var/run:/var/run:ro          # Mount host /var/run (read-only)
      - /sys:/sys:ro                  # Mount host /sys (read-only)
      - /var/lib/docker/:/var/lib/docker:ro  # Mount Docker data directory (read-only)
      - /dev/disk/:/dev/disk:ro       # Mount host disk info (read-only)
    networks:
      - monitoring-network
    restart: unless-stopped


# Persistent volumes
volumes:
  strapi-data: # Persistent storage for the strapi database
  prometheus_data:  # Volume for storing Prometheus data

# Network configurations
networks:
  app-network: # App network
    name: App
    driver: bridge
  backend-network: # Backend network
    name: Backend
    driver: bridge
  database-network: # Database network
    name: Database
    driver: bridge
  cert-network: # Cert network
    name: Cert
    driver: bridge
  monitoring-network: # monitoring-network
    name: monitoring-network
    driver: bridge

```

FIGURE 55 – Docker Compose partie 2

D.4 NGINX

```
# Server block for direct IP address
server {
    listen 80;  # Listen on port 80 for HTTP connections
    server_name [redacted];  # Respond to requests for this IP address
    # Redirect all requests to the HTTPS version of aniss.dev
    location / {
        return 301 https://aniss.dev$request_uri;
    }
}

# HTTP server block for strapi.aniss.dev
server {
    listen 80;  # Listen on port 80 for HTTP connections
    listen [::]:80;  # Listen on IPv6 as well

    server_name strapi.aniss.dev;  # Respond to requests for this domain
    server_tokens off;  # Disable nginx version display in server responses
    # Handle ACME challenges for Let's Encrypt SSL certification
    location ~ /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }
    # Redirect all HTTP requests to HTTPS
    location / {
        return 301 https://strapi.aniss.dev$request_uri;
    }
}


# HTTP server block for blog.aniss.dev
server {
    listen 80;  # Listen on port 80 for HTTP connections
    listen [::]:80;  # Listen on IPv6 as well

    server_name blog.aniss.dev;  # Respond to requests for this domain
    server_tokens off;  # Disable nginx version display in server responses

    # Handle ACME challenges for Let's Encrypt SSL certification
    location ~ /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    # Redirect all HTTP requests to HTTPS
    location / {
        return 301 https://blog.aniss.dev$request_uri;
    }
}

# HTTPS server block for blog.aniss.dev
server {
    listen 443 ssl;  # Listen on port 443 for HTTPS connections
    listen [::]:443 ssl;  # Listen on IPv6 as well
    http2 on;  # Enable HTTP/2 protocol

    server_name blog.aniss.dev;  # Respond to requests for this domain

    # SSL certificate configuration
    ssl_certificate /etc/nginx/certs/live/aniss.dev/fullchain.pem;
    ssl_certificate_key /etc/nginx/certs/live/aniss.dev/privkey.pem;

    # Proxy requests to Next.js application
    location / {
        proxy_pass http://blogFront:3000;  # Forward requests to Next.js container
        proxy_http_version 1.1;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Server $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $http_host;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_pass_request_headers on;
    }


}


# HTTPS server block for strapi.aniss.dev
server {
    listen 443 ssl;  # Listen on port 443 for HTTPS connections
    listen [::]:443 ssl;  # Listen on IPv6 as well
    http2 on;  # Enable HTTP/2 protocol

    client_max_body_size 100M;  # Set maximum allowed size of the client request body

    server_name strapi.aniss.dev;  # Respond to requests for this domain

    # SSL certificate configuration
    ssl_certificate /etc/nginx/certs/live/aniss.dev/fullchain.pem;
    ssl_certificate_key /etc/nginx/certs/live/aniss.dev/privkey.pem;

    # Proxy requests to Strapi application
    location / {
        proxy_pass http://strapi:1337;  # Forward requests to Strapi container
        proxy_http_version 1.1;
        # Set various proxy headers
        proxy_set_header X-Forwarded-Host $host;  # Set the original host header
        proxy_set_header X-Forwarded-Server $host; # Set the original server header
        proxy_set_header X-Real-IP $remote_addr; # Set the original client IP address
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for; # Set the original forwarded for header
        proxy_set_header X-Forwarded-Proto $scheme; # Set the original protocol
        proxy_set_header Host $http_host; # Set the original host header
        proxy_set_header Upgrade $http_upgrade; # Set the upgrade header
        proxy_set_header Connection "Upgrade"; # Set the connection header
        proxy_pass_request_headers on; # Pass the original request headers
    }

}
```

FIGURE 56 – Nginx configuration partie 1

```
# HTTP server block for aniss.dev and www.aniss.dev
server {
    listen 80;  # Listen on port 80 for HTTP connections
    listen [::]:80;  # Listen on IPv6 as well

    server_name aniss.dev www.aniss.dev;  # Respond to requests for these domains
    server_tokens off;  # Disable nginx version display in server responses
    # Redirect all HTTP requests to HTTPS
    location / {
        return 301 https://$host$request_uri;
    }

    # Handle ACME challenges for Let's Encrypt SSL certification
    location ~ /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }
}

# HTTPS server block for www.aniss.dev and aniss.dev
server {
    listen 443 default_server ssl;  # Listen on port 443 for HTTPS connections, set as default
    listen [::]:443 ssl;  # Listen on IPv6 as well
    http2 on;  # Enable HTTP/2 protocol

    server_name aniss.dev www.aniss.dev;  # Respond to requests for these domains

    # SSL certificate configuration
    ssl_certificate /etc/nginx/certs/live/aniss.dev/fullchain.pem;
    ssl_certificate_key /etc/nginx/certs/live/aniss.dev/privkey.pem;

    root /usr/share/nginx/html;  # Set the root directory for serving files
    index index.html;  # Define the default index file

    # Serve static files or return 404 if not found
    location / {
        try_files $uri $uri/ =404;
    }


}

# HTTP server block for adminer.aniss.dev
server {
    listen 80;
    listen [::]:80;

    server_name adminer.aniss.dev;
    server_tokens off;

    # Handle ACME challenges for Let's Encrypt SSL certification
    location ~ /.well-known/acme-challenge/ {
        root /var/www/certbot;
    }

    # Redirect all HTTP requests to HTTPS
    location / {
        return 301 https://adminer.aniss.dev$request_uri;
    }
}

# HTTPS server block for adminer.aniss.dev
server {
    listen 443 ssl;
    listen [::]:443 ssl;
    http2 on;

    server_name adminer.aniss.dev;

    # SSL certificate configuration
    ssl_certificate /etc/nginx/certs/live/aniss.dev/fullchain.pem;
    ssl_certificate_key /etc/nginx/certs/live/aniss.dev/privkey.pem;

    # Proxy requests to Adminer
    location / {
        proxy_pass http://strapiAdminer:8080;
        proxy_http_version 1.1;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Server $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $http_host;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_pass_request_headers on;
    }


}

```

FIGURE 57 – Nginx configuration partie 2

D.5 WORKFLOW GITHUB ACTION

```yaml
# Workflow name
name: Test, Build, Push, and Deploy Docker Image

# Trigger the workflow on push to the main branch
on:
  push:
    branches:
      - main

# Environment variables used throughout the workflow
env:
  REGISTRY: ghcr.io # GitHub Container Registry
  IMAGE_NAME: ${{ github.repository_owner }}/${{ github.event.repository.name }} # Format: owner/repo-name

# Define the jobs to run tests, then build and push the Docker image, and finally deploy
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: "20" # Adjust this to your project's Node.js version

      - name: Install dependencies
        run: yarn install

      - name: Run tests
        run: yarn test

  build-and-push:
    needs: test # This job will only run if the test job succeeds
    runs-on: ubuntu-latest
    permissions:
      contents: read # Permission to read repository contents
      packages: write # Permission to write packages (for pushing Docker images)

    steps:
      # Step 1: Check out the repository code
      - name: Checkout repository
        uses: actions/checkout@v3

      # Step 2: Convert the repository name and owner to lowercase
      - name: Lowercase the repo name and owner
        run: |
          echo "REPO_LOWER=${GITHUB_REPOSITORY,,}" >>${GITHUB_ENV}
          echo "OWNER_LOWER=${GITHUB_REPOSITORY_OWNER,,}" >>${GITHUB_ENV}

      # Step 3: Log in to the GitHub Container Registry
      - name: Log in to the Container registry
        uses: docker/login-action@v2
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }} # GitHub username running the workflow
          password: ${{ secrets.GITHUB_TOKEN }} # Automatically provided GitHub token

      # Step 4: Extract metadata for Docker
      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@v4
        with:
          images: ${{ env.REGISTRY }}/${{ env.OWNER_LOWER }}/${{ github.event.repository.name }}

      # Step 5: Build and push the Docker image
      - name: Build and push Docker image
        uses: docker/build-push-action@v4
        with:
          context: . # Build context is the root of the repository
          push: true # Push the image to the registry
          tags: | # Tags for the Docker image
            ${{ env.REGISTRY }}/${{ env.OWNER_LOWER }}/${{ github.event.repository.name }}:latest
            ${{ env.REGISTRY }}/${{ env.OWNER_LOWER }}/${{ github.event.repository.name }}:${{ github.sha }}
          labels: ${{ steps.meta.outputs.labels }} # Labels extracted in the previous step

  deploy:
    needs: build-and-push # This job will only run if the build-and-push job succeeds
    runs-on: ubuntu-latest

    steps:
      - name: Deploy to VPS
        env:
          VPS_SSH_PRIVATE_KEY: ${{ secrets.VPS_SSH_PRIVATE_KEY }}
          VPS_HOST: ${{ secrets.VPS_HOST }}
          VPS_USER: ${{ secrets.VPS_USER }}
          VPS_SSH_PORT: ${{ secrets.VPS_SSH_PORT }}
        run: |
          echo "$VPS_SSH_PRIVATE_KEY" > vps_ssh_key
          chmod 600 vps_ssh_key
          ssh -i vps_ssh_key -p $VPS_SSH_PORT -o StrictHostKeyChecking=no $VPS_USER@$VPS_HOST '
            cd /home/torka/monSite &&  docker-compose up -d
          '
          rm vps_ssh_key

```

FIGURE 58 – Workflow GIthub Action