[redacted]

[redacted]

Date limite de rendu : vendredi 26 avril 23h00

Vous rendrez une archive ZIP contenant tout votre code, ainsi qu'un rapport (succinct, 4-5 pages maxi) présentant votre application, les choix techniques, et les difficultés rencontrées pendant sa réalisation.

[redacted]

[redacted]

[redacted]

- [redacted]
- [redacted]
- Pour le côté temps réel, un WebSocket sera utilisé. Un module serveur Nuxt sera mis en place pour réaliser un serveur WebSocket, et l'application s'y connectera afin d'être notifiée en temps réel des réponses apportées.
- [redacted] L'application initialisera le schéma de la base de données au premier démarrage.
- L'application devra être packagée dans un Docker, et doit fournir un fichier docker-compose.yml pour pouvoir être lancée via un simple docker compose up

[redacted]

[redacted]

Les utilisateurs doivent créer un [redacted] à l'application

[redacted] Un compte par défaut “admin” (avec le mot de passe “admin”) sera créé automatiquement par la plate-forme lorsqu'elle est lancée pour la première fois.

[redacted] et changer leur mot de passe depuis un bouton prévu à cet effet.

[redacted]

[redacted]

[redacted] un bouton “Nouveau sujet” permettant [redacted]

Les sujets doivent être paginés, à raison de 20 sujets par page.

[redacted]

[redacted]

Les messages doivent être paginés, à raison de 20 messages par page.

[redacted]

L’application doit s’actualiser en temps réel : si un utilisateur poste un message dans un sujet, ou un nouveau sujet, la liste des sujets au sein d’un forum doit s’actualiser immédiatement. De la même manière, si je suis en train de consulter un sujet et qu’un autre utilisateur poste un nouveau message, le nouveau message posté doit immédiatement apparaître.

[redacted]

Eléments techniques pour la réalisation

Mise en place d'un serveur WebSocket

Installation et activation

Le serveur node Nuxt doit pouvoir héberger un serveur WebSocket directement. Pour ce faire, nous utiliserons directement la fonction WebSocket intégrée à Nitro, le module qui s'occupe du SSR.

Pour cela, activez le module à votre fichier nuxt.config.ts :

```
export default defineNuxtConfig({
  // …
  modules: [ /* … autres modules … */ ],
  nitro: {
    experimental: {
      websocket: true
    }
  }
})
```

Utilisation

Nous allons créer une route "/_ws", qui pointera vers notre point d'entrée WebSocket. Pour cela, nous allons déclarer une route directement côté serveur (pas gérée par le routeur de Nuxt, mais directement côté serveur HTTP).

Créez pour cela le fichier `server/routes/_ws.ts` et ajoutez-y ce contenu :

```
export default defineWebSocketHandler({
  open(peer) {
    console.log("[ws] open", peer);
  },

  message(peer, message) {
    console.log("[ws] message", peer, message);
    if (message.text().includes("ping")) {
      peer.send("pong");
    }
  },

  close(peer, event) {
    console.log("[ws] close", peer, event);
```

```
  },

  error(peer, error) {
    console.log("[ws] error", peer, error);
  },
});
```

Cette route a pour effet :

- Fonction “`open`”: Appelée lorsqu’un client se connecte au serveur WebSocket
- Fonction “`message`” : Appelée lorsqu’un client envoie un message au serveur. Dans cet exemple, on log le message reçu, et on renvoie “pong” si le client envoie “ping”
- Fonction “`close`”: Appelée lorsqu’un client se déconnecte du serveur
- Fonction “`error`”: Appelée lorsqu’une erreur survient

Pour pouvoir diffuser un message à tous les utilisateurs connectés, il faudra donc stocker tous les clients (peer) connectés (lors de la fonction “open”), et leur envoyer un message tant que l’on n’a pas eu d’événement “close” ou “error” sur le peer en question.

Côté client, nous allons donc créer une page `pages/socket.vue` permettant de tester la connexion à notre WebSocket :

```
<script setup>
let ws
const messages = ref([])

// Fonction de connexion au serveur WebSocket
const connect = async () => {
    // En HTTP, le protocole ws:// est utilisé. En HTTPS, il est
nécessaire
    // d'utiliser le protocole wss://.
    const isSecure = location.protocol === "https:";
    const url = (isSecure ? "wss://" : "ws://") + location.host +
"/_ws";
    if (ws) {
        // Déjà connecté, on ferme la connexion existante
        ws.close();
    }

    // Connexion au serveur
    console.log("Connexion à", url, "...");
    ws = new WebSocket(url);

    // Ajout d'un listener pour être notifié lorsque le serveur
```

```
    // nous envoie un message
    ws.addEventListener("message", (event) => {
        const message = event.data
        console.log(message);
        messages.value.push(message)
    });

    // On attend d'être connecté. L'objet WebSocket natif n'utilise
    // pas les promesses, donc on triche un peu pour pouvoir
utiliser
    // await sur l'étape de connexion.
    await new Promise((resolve) => ws.addEventListener("open",
resolve));
    console.log("ws connecté!");
};

const ping = () => {
    console.log("ws envoi ping");
    ws.send("ping");
};
</script>

<template>
    <div>Messages: {{ messages }}</div>
    <button @click="connect">Connecter</button>
    <button @click="ping">Ping</button>
</template>
```