[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 compte et s'authentifier à l'application pour pouvoir créer des sujets ou poster des messages. Toutefois, il n'est pas nécessaire d'être authentifié pour consulter les forums, sujets et messages postés. Deux niveaux de droits existent, un utilisateur simple, et un administrateur. 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]

Toujours sur la page du forum, un bouton "Nouveau sujet" permettra de créer un nouveau sujet. L'utilisateur doit alors saisir un titre, et un premier message pour le sujet (champ texte multi-lignes). Ce bouton n'est accessible que si l'utilisateur est connecté, sinon un message l'invitera à se créer un compte ou à se connecter.

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

Lorsque l'on sélectionne un sujet, on doit arriver sur une page qui liste tous les messages du sujet, par ordre de date de message croissant. Le premier message (initial) est donc en haut de la page 1, le dernier message posté en réponse sera en bas de la dernière page. Les utilisateurs connectés peuvent cliquer sur un bouton Répondre pour pouvoir apporter une réponse au sujet, sous la forme d'un champ texte, comme pour le message initial à la création d'un sujet.

Les messages doivent indiquer pour chacun le nom de l'utilisateur qui l'a posté, la date à laquelle le message a été posté, et évidemment le message en lui-même.

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

Un utilisateur doit pouvoir modifier un de ses propres messages via un bouton "Modifier" présent au niveau de chaque message qu'il aura posté. Un administrateur peut modifier les messages de tous les utilisateurs, supprimer un message posté, ou supprimer complètement un sujet et tous ses messages.

Un espace administrateur permettra de créer un nouveau compte administrateur, et de créer, renommer ou supprimer un forum. Lorsqu'un forum est supprimé, tous ses sujets et messages doivent l'être également.

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.

Fonctionnalités bonus

Ces fonctionnalités ne sont pas obligatoires mais si réalisées peuvent donner un complément de points.

- Possibilité de citer un message : Un bouton "Citer" présent sur les messages permet de citer un message existant et d'y répondre. Dans la réponse, le message cité apparaît en premier, puis le message saisi par l'utilisateur.
- Avatar : Les utilisateurs peuvent uploader une image en guise d'avatar, affichée à côté du nom sur chacun des messages postés par celui-ci.
- Verrouiller un sujet : Un bouton "Verrouiller", disponible qu'aux administrateurs, permet de verrouiller un sujet et empêcher tous les utilisateurs d'y apporter de nouveaux messages.
- Etat lu/non-lu : Un indicateur doit montrer quels sont les sujets et messages que l'utilisateur n'a pas encore lu depuis sa dernière visite.

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>
```

Pool de connexion MySQL

Pour pouvoir mieux tirer parti de la connexion MySQL et éviter de dupliquer le code, nous allons pouvoir définir un modèle central et le réutiliser.

Créez un fichier “`server/utils/mysql.ts`”, contenant le code de connexion à MySQL, et exportant la connexion :

```
import mysql from 'mysql2/promise'
import bluebird from 'bluebird'
import type { EventHandler, EventHandlerRequest } from 'h3'

export const defineWrappedResponseHandler = <T extends
EventHandlerRequest, D>(
    handler: EventHandler<T, D>
): EventHandler<T, D> =>
    defineEventHandler<T>(async event => {
        try {
            const connection = await mysql.createConnection({
```

```
            host: 'localhost',
            user: 'root',
            password: 'root',
            database: 'movies',
            Promise: bluebird,
        })
        event.context.mysql = connection

        const response = await handler(event)

        event.context.mysql.end()
        return response
    } catch (err) {
        // Error handling
        return { err }
    }
})
```

Il sera ensuite nécessaire d'utiliser `defineWrappedResponseHandler` au lieu de `defineEventHandler` dans vos routes API serveur. Par exemple, dans un fichier `server/api/movies.get.ts` :

```
export default defineWrappedResponseHandler(async (event) => {
    const db = event.context.mysql
    const [rows, fields] = await db.execute("SELECT * FROM movies")
    return {
        movies: rows
    }
});
```

Utilisation de Vuetify

Vous pouvez utiliser Vuetify pour améliorer l'aspect graphique de votre forum. Tout autre framework de votre choix sera également accepté.

Pour installer Vuetify, suivez les instructions présentes dans la documentation de Vuetify 3, à la section dédiée pour Nuxt :
https://vuetifyjs.com/en/getting-started/installation/#using-nuxt-3

La documentation vous sera également utile pour consulter tous les composants du framework et connaître leur fonctionnement ;-)