TD4
VueJS 3 - SPA / Router

Introduction

Dans ce TD vous allez créer un jeu de morpion. L'API est hébergée sur https://morpion-api.edu.netlor.fr

Le TD doit être effectué en binôme.

Liste des routes disponibles :

Verbe	Route	Body	Description
GET	/api/ping	-	Permet de vérifier que l'API est accessible et que vous arrivez à l'atteindre

POST	/api/apikeys	{
		"name": …,
		"email": …
		}
			Permet d'obtenir une API KEY nécessaire pour pouvoir utiliser le reste des routes. Retourne vos données, incluant l'API KEY associée à votre compte.

GET	/api/profile	-	Permet de retourner les informations de l'utilisateur courant

PUT	/api/profile	{
		"name": …,
		"email": …
		}
			Permet de modifier votre nom et votre adresse email

POST	/api/games	{}	Permet de créer une nouvelle partie. Elle retourne les données de la partie, dont un code qui permet à l'adversaire de rejoindre votre partie
PATCH	/api/games/:code/join	-	Permet de rejoindre une partie grâce à son code (dans l'url)
GET	/api/games/:id	-	Permet de récupérer les informations d'une partie grâce à l'id de celle-ci, fourni dans l'url. Côté backend, la route est prévue pour fonctionner selon le principe du long-polling. Elle met en attente la requête du client pendant maximum 30 secondes, et déclenche une réponse dès qu'un changement a lieu sur la partie (ex : un adversaire rejoint la partie, un joueur a joué, la partie est terminée). Au bout des 30 secondes, la route retournera l'état courant de la partie.
PATCH	/api/games/:id/play/:row/:col	-	Permet de jouer une cellule du morpion. L'url attend : * :id, l'id de la partie * :row, la ligne ciblée * :col, la colonne ciblée

Fonctionnalités attendues

Maquette non “contractuelle” :)

L’application que vous devez développer doit vous permettre :
* de créer une nouvelle partie,
* de rejoindre une nouvelle partie et jouer cette partie,
* d’éditer les informations de votre profil

Une première vue doit être proposée à l’utilisateur permettant d’accéder à ces trois choix.

L’utilisateur doit pouvoir modifier son profil à travers un formulaire dédié.

En créant une partie, l’utilisateur est redirigé sur la vue de la partie. La partie reste en attente tant qu’aucun adversaire n’a rejoint la partie. La vue doit faire apparaître le code de celle-ci pour que l’utilisateur puisse le communiquer à son adversaire.

L’adversaire peut alors rejoindre la partie en saisissant le code.

Dès qu'un adversaire rejoint la partie, celle-ci peut commencer. L'API détermine aléatoirement lequel des deux joueurs commencera.

Un indicateur visuel doit permettre d'identifier le joueur dont c'est le tour.

L'actualisation de la partie passe par une interaction avec les websockets mis à disposition par l'API (https://morpion-api.edu.netlor.fr/websockets).

L'API détecte la fin de partie (soit match nul, soit victoire d'un des deux joueurs).

À la fin de la partie, le résultat final doit être affiché. Proposer un bouton permettant de revenir au menu principal.

Exercice 1

Pour commencer vous devez créer une API KEY. Ouvrez un logiciel permettant de tester des API tel que POSTMAN (https://www.postman.com/downloads/).

* Lancez le logiciel
* Effectuez une requête POST sur la route `/api/apikeys` en donnant le body attendu par l'API (name et email).
* Étudiez les résultats, vous devriez voir une "key".
* Copiez-la, elle vous servira pour le reste des exercices.
* Chaque individu de votre binôme doit créer son API KEY pour pouvoir être identifié distinctement par l'API.

Exercice 2

Dans cet exercice vous devez initier une nouvelle application SPA avec VueJS.

* Assurez-vous que votre version de node est à jour (version 18 ou +)
* Créez l'application (`npm create vue@latest`)
* À la question demandant si vous souhaitez installer vue-router, répondez oui
* Idem pour ESLint (permet d'assurer la qualité du code)
* Suivez les instructions pour installer les paquets et lancer l'application
* Installez axios
* Observez la structure du code généré par VueJS, le point d'entrée, le main, les composants le router,
* Supprimez tout ce dont vous n'avez pas besoin (styles, icons, contenu du routeur, views, components), pour partir "au propre"

Exercice 3

Cet exercice prévoit la configuration d'Axios.

* Dans le dossier `src`, créez un dossier `api/` et sous ce dossier un fichier `index.js`
* Dans ce fichier, exportez par défaut une instance d'axios dont le fichier de configuration prévoit :
  * une url de base (`baseURL`), qui pointe sur https://morpion-api.edu.netlor.fr
  * une propriété `headers`, dont la valeur est un objet qui propose deux headers :
    * `Content-Type: 'application/json'`. Cela permet d'indiquer à l'API que l'on va communiquer avec elle en JSON
    * `Authorization: 'key=[votre_api_key]'`. Permet d'authentifier votre client auprès de l'API. Chacun des deux individus du binôme doit indiquer dans son code sa propre API KEY.

Si vous êtes seul sur ce projet, vous pouvez créer deux API KEY différentes et modifier votre application de façon à ce que l'utilisateur saisisse son API KEY avant de pouvoir créer/rejoindre/jouer une partie. Il faudra alors rendre configurable dynamiquement votre instance d'axios. Vous pourrez alors simuler les interactions entre les deux joueurs avec deux onglets différents de votre navigateur

Exercice 4

Cet exercice permet de gérer la page d'accueil, donnant accès au sommaire de l'application.

* Dans le router de l'application, créez une route pour la page d'accueil, `/home`
* Prévoyez un composant `Home.vue` dans le répertoire `src/views`, qui propose les trois accès principaux
* Le template du composant principal (`App`) ne doit contenir finalement que le `RouterView`

Exercice 5

Cet exercice va permettre d'éditer les informations de profil de l'utilisateur.

* Dans le router de l'application, prévoyez une nouvelle route `/profile`
* Créez une nouvelle vue `Profile.vue` dans le dossier `src/views` qui proposera un formulaire permettant d'éditer les informations de votre utilisateur
* Prévoyez une data représentant votre utilisateur, par exemple `user`, qui vaut `null` pour le moment,
* Dans le hook `beforeRouteEnter` du composant, faites un appel Ajax sur la route `GET /api/profile` pour pouvoir alimenter les informations de user
* Dans le template du composant, bindez les champs `name` et `email` sur les propriétés correspondantes de votre user (`v-model`)
* Au clic sur le bouton Enregistrer, lancez un appel Ajax sur la route `PUT /api/profile`. Le body de cette requête sera votre user
* Assurez-vous que la requête s'est bien passée. S'il y a une erreur remontée par le serveur, faites en sorte de l'afficher (ex : Le format de l'adresse email est incorrect). Attention : il peut y avoir plusieurs erreurs à la fois.
* Proposez un bouton (croix, lien, …) qui permet à l'utilisateur de revenir à l'accueil
* Adaptez la vue `Home.vue` pour qu'un clic sur le bouton MON PROFIL vous redirige vers cette route

Exercice 6

Cet exercice permet de créer une partie.

* Dans le router de l'application, prévoyez une route `/games/:id`. Cette route doit être nommée (`name: "game"`)
* Dans votre vue Home.vue, ajoutez une méthode createGame() dans la section methods
* Cette fonction doit créer une nouvelle partie au moyen d'un appel ajax POST sur /api/games
* Observez les résultats de la requête Ajax afin d'en extraire l'id
* Redirigez l'utilisateur vers /games/:id où :id sera l'id de votre partie.

Exercice 7

À présent vous devez créer la vue qui accueille la partie.

* Créez un composant Game.vue dans src/views
* Prévoyez dans vos data un objet qui représente votre partie. Au départ, cet objet vaut null

* Dans le hook beforeRouteEnter de votre composant, réalisez un appel Ajax GET sur /api/game/:id où id est l'id de votre partie. Affectez le résultat à votre data représentant la partie.
* Toujours dans le beforeRouterEnter, après avoir récupéré les informations de la partie, déclenchez l'actualisation automatique de la partie
  * Pour cela vous devez initier une connexion avec les websockets de l'API (https://morpion-api.edu.netlor.fr/websockets), dans une méthode "waitForOpponentMove"
  * Après avoir ouvert la connexion, envoyer un message "{action: 'connect', game_id: 'id_de_la_partie', player_id: 'id_de_l_utilisateur_connecté'}"
  * Dans la callback de traitement des messages émis par les WS, traitez les différents cas :
    * opponent-join : votre adversaire vient de rejoindre la partie,
    * opponent-play : votre adversaire vient de jouer une cellule,
    * opponent-quit : votre adversaire a quitté la partie
* Vous pouvez afficher votre nom
* Affichez le code de la partie, afin de pouvoir le communiquer à votre adversaire
* Adaptez l'affichage de votre composant selon que votre adversaire a déjà rejoint ou pas la partie. Dès qu'il rejoint la partie, la propriété opponent sera remplie par le serveur et vous pourrez afficher ses informations dans la vue.
* L'API vous donne également l'information du prochain joueur : next_player_id. Cela vous permettra d'indiquer à l'écran lequel doit commencer
* Tant qu'aucun opponent n'est indiqué dans la partie, affichez un message qui indique à l'utilisateur que la partie est en attente d'un adversaire
* Dès qu'un opponent est disponible, affichez la grille de jeu

Exercice 8

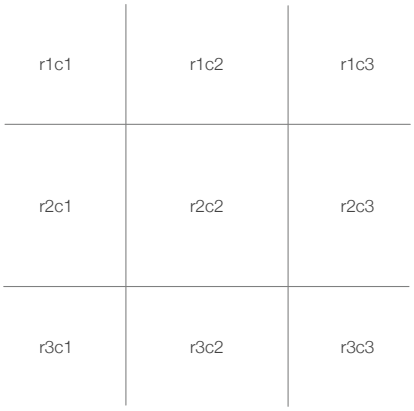
On souhaite à présent permettre à l'adversaire de rejoindre la partie.

* Créez une nouvelle route dans le router : /join
* Créez un composant associé à cette route, dans src/views : Join.vue
* Ce composant doit proposer un champ permettant de renseigner le code de la partie que l'utilisateur souhaite rejoindre
* Bindez une propriété de vos data sur le champ du formulaire
* Au clic sur le bouton Rejoindre, faites un appel Ajax sur la route /api/games/:code/join
* Affichez les erreurs si le serveur en retourne (ex : si le code saisi n'existe pas côté serveur)
* Si tout se passe bien, l'API retourne les informations de la partie, dont son id
* Redirigez l'utilisateur vers la partie (/games/:id)

Exercice 9

À ce stade, les deux utilisateurs sont sur la page de la partie. Celle-ci peut commencer.

* Créez la méthode play dans la section methods du composant Game
* Faites en sorte qu'un clic sur une cellule de la grille appelle cette méthode
* La méthode doit réaliser un appel ajax sur la route PATCH /api/games/:id/play/:row/:col
* La grille est numérotée ainsi :



* :row correspond au numéro de la ligne
* :col correspond au numéro de la colonne
* Assurez vous que si l'API renvoie une erreur (par exemple la cellule ciblée a déjà été jouée), l'action de l'utilisateur ne soit pas prise en compte et qu'un message d'erreur apparaisse (ou qu'un effet visuel apparaisse sur la cellule cliquée)
* Si un changement est détecté par la méthode waitForOpponentMove, la grille s'actualisera automatiquement pour faire apparaître le coup joué par votre adversaire
* Si le statut de la partie passe à 2, alors la partie est terminée. Remplacez alors la grille par un message indiquant la fin de partie. Si un gagnant a été détecté par l'API, le champ winner_id de la partie retournée par l'API contiendra l'id du winner. Vous pouvez afficher son nom le cas échéant.
* À noter : la route /api/games/:id/play/:row/:col renvoie la partie en valeur de retour de l'appel Ajax.
* La vue doit proposer un bouton (ou un lien) permettant de revenir à l'accueil, à l'issue de la partie.
* Faites en sorte qu'en quittant la partie (ex : en revenant au menu principal), la connexion aux websockets soit explicitement fermée

Exercice 10

Les vues Profile et Join partagent un mécanisme commun lié à l'affichage d'erreurs. Faites en sorte que l'affichage des erreurs soit piloté par un composant dédié, de manière à rendre générique l'affichage des erreurs.