


R4A.10 - Compléments Web (vuejs)

TP n°2 : Heroes & Vilains - upgrade

Détails

Écrit par stéphane Domas

Catégorie : R4A.10 - Compléments web (vuejs) (/index.php/menu-cours-s4/menu-mmi4web)

 Publication : 18 décembre 2020

 Affichages : 1377

Préambule

- Le but de ce TP est de d'améliorer l'application SPA "Heroes & Vilains", commencée dans le TP 1, en ajoutant les mécanismes vuejs abordés dans les premiers TD.

1°/ Modulariser le store

- Dans les démonstrations des TDs, on utilise un store modularisé.
- Pour cet exercice, il faut refactoriser le store afin qu'il soit modulaire, avec 3 modules :
 - un pour la gestion des données générale de l'application
 - un pour le traitement des erreurs (cf. exercice 2),
 - un pour la gestion de la phrase secrète (cf. exercice 3),
 - un pour la gestion des utilisateurs héros (cf. exercice 5),
- Cela implique bien entendu de modifier toutes les utilisations de mappers dans les composants déjà écrits.

2°/ Traitement des erreurs

- L'objectif de cet exercice est de traiter tous les cas d'erreur de l'application en utilisant les principes vu en, à savoir grâce à une boîte de dialogue instanciée comme sous-composant du composant racine App.
- Les situations d'erreur sont notamment celles où les appels à l'API produit une erreur.

3°/ La phrase secrète

- Pour accéder à une organisation, il faut taper sa phrase secrète.
- Dans le TP 1, cette phrase secrète est envoyée via l'URL lors de certaines requêtes, ce qui n'est pas très sécurisé puisque quelqu'un qui voit l'URL connaît le secret.
- Le but de cet exercice est simplement d'utiliser le principe des intercepteurs axios vus en TD 2 pour que :
 - si la phrase secrète existe dans le store, on l'envoie dans une entête http nommé org-secret.
 - sinon, on n'envoie pas cette entête.
- Le seul inconvénient de cette procédure est qu'elle envoie l'entête pour toutes les requêtes, et pas seulement pour celles où il faut envoyer le secret. Mais ce n'est pas très important.

4°/ Gardes de routage

- L'objectif de l'exercice est de mettre en place :
 - une restriction d'accès aux routes qui affichent des composants privilégiés, à savoir ceux qui nécessitent que l'utilisateur ait la bonne phrase secrète pour récupérer les données à afficher.
 - une redirection vers le composant de saisie de la phrase secrète dès que l'on veut suivre une route privilégiée alors que l'on est pas authentifié.
 - une redirection vers le composant d'accueil si l'on suit une route non définie.
- Pour cela, il suffit d'utiliser les principes vus en TD 1.

5°/ Accès personnel pour les héros

5.1°/ Authentification utilisateur via login/mot de passe

- L'API utilisée dans les TPs permet également de gérer une collection d'utilisateurs, représentant le compte personnel d'un héros.
- Le schéma mongodb de la collection Users est le suivant :

```
1 | let UserSchema = new Schema({
2 |   login: {type: String, required: true, minLength:2},
3 |   password: {type: String, required: true, minLength:2},
4 |   hero: {type: Schema.Types.ObjectId, ref: 'Heroe'},
5 | }
```

- Comme on peut le constater, un utilisateur est répertorié avec un login, un mot de passe et l'identifiant d'un héros existant.
- Dans la base de données, il y a déjà 4 utilisateurs avec les informations suivantes :
 - login : superdupond, mot de passe : azer, héros associé : super dupond
 - login : chatounette, mot de passe : azer, héros associé : chatounette
 - login : maddog, mot de passe : azer, héros associé : mad dog
 - login : supertutu, mot de passe : azer, héros associé : super tutu
- Pour s'authentifier, l'API propose une route `/authapi/auth/signin`
 - méthode POST, les données jointes sont un objet au format `{ login: ..., password: ...}`.
 - en cas de succès, la réponse est un objet au format `{err: 0, status: 200, data: { name : login_user, xsrfToken: ..., refreshToken: ... } }`. De plus, un cookie JWT est renvoyé afin de vérifier l'authentification pour les requêtes suivantes.
 - en cas d'échec, la réponse est un objet au format `{err: XXX, status: 400, data: 'message erreur' }`, avec `XXX > 0`

NB 1 : vous pouvez stocker le token xsrf où vous voulez, par exemple dans le store, ou bien dans le local storage du navigateur.

NB 2 : l'API supporte le rafraîchissement de token xsrf/JWT mais ce n'est pas utilisé dans ce TP.

- Après authentification, il est possible de récupérer les informations d'un utilisateur, avec la route : `/authapi/user/getuser/:login`
 - méthode GET, le login étant indiqué comme dernier élément de la route.
 - en cas de succès, le champ data de la réponse est un objet contenant le login, le mot de passe, et un objet décrivant le héros associé, comme celui que l'on récupère avec la route `/herocorp/heroes/getbyid/:id`.
 - en cas d'échec, la réponse est un objet au format `{err: XXX, status: 400, data: 'message erreur' }`, avec `XXX > 0`
- ATTENTION ! Si l'API ne reçoit pas le cookie JWT et/ou le token xsrf dans les entêtes, ou bien si la valeur du xsrf ne correspond à celle stockée dans le JWT, suivre cette route est forcément un échec.
- Le cookie JWT est envoyé automatiquement, mais le token xsrf doit être renvoyé explicitement dans une entête http nommée : `x-xsrf-token`.
- Pour envoyer cette entête, vous pouvez soit réutiliser l'intercepteur axios que vous avez écrit pour envoyer le secret, soit créer une requête axios spécifique.
- L'objectif de l'exercice est d'ajouter un composant permettant à un héros de s'authentifier auprès de l'API.

- Pour cela, il suffit de partir du code de démonstration du TD 2, qui contient déjà tout ce qui est nécessaire pour cette authentification, notamment la gestion des tokens dans le localStorage, ainsi que le rafraîchissement du JWT.

ATTENTION ! Dans le TP 1, toutes les routes d'accès à l'API commencent par /herocorp. Comme les routes qui concernent l'authentification commencent par /authapi, il faut que votre instance d'axios puisse suivre ces 2 types de routes.

5.2°/ Mise à jour d'un héros

- Dans le TP1, il est possible d'ajouter et modifier un héros à partir du composant affichant les membres d'une équipe.
- Cette fonctionnalité n'est normalement accessible que si on a fourni à l'API la phrase secrète de l'organisation qui possède l'équipe.
- L'API permet également à un utilisateur héros de modifier ses informations à partir du moment où il est authentifié.
- Pour cela, l'API propose la route : /herocorp/heroes/authupdate.
- Cette route fonctionne presque de la même façon que la route update utilisée dans le TP 1 : mêmes type de requête, données fournies, mais en vérifiant si l'authentification est correcte (comme pour /getuser) au lieu de se baser sur la phrase secrète.
- L'objectif de l'exercice est d'ajouter la possibilité de ;
 - récupérer les informations d'un utilisateur (cf. exo 1) pour en tirer les informations du héros associé.
 - modifier ces informations en utilisant la route décrite ci-dessus.

5.3°/ Enregistrement

- Dans la démonstration du TD 2, il y a un exemple d'enregistrement utilisateur, protégé via captcha google.
- En s'inspirant de cet exemple, ajouter un composant permettant d'enregistrer un nouvel utilisateur, associé à un héros.
- La route de l'API permettant l'enregistrement est : /authapi/user/register
 - méthode POST, les données jointes sont {login: ..., password: ..., hero: ..., captchaToken: ...}.
 - Le champ hero doit contenir le nom public d'un héros existant (cf démo du TD2)
- Pour information, cette route renvoie une erreur si :
 - le login est déjà pris,
 - le nom du héros fourni n'existe pas,
 - le nom du héros existe mais est déjà associé à un utilisateur.

ATTENTION : pour que le captcha fonctionne correctement, il faut fournir une clé à l'API lors de la requête à la route. Dans la démonstration du TD 2, cette clé se trouve dans le fichier src/commons/config.js et est nommé captchaSiteKey. Pour ce TP, vous devez changer la valeur de cette clé et mettre : '6LdzWJkkAAAAAOG1QjA_LmULztIgSqmGokocys0x'