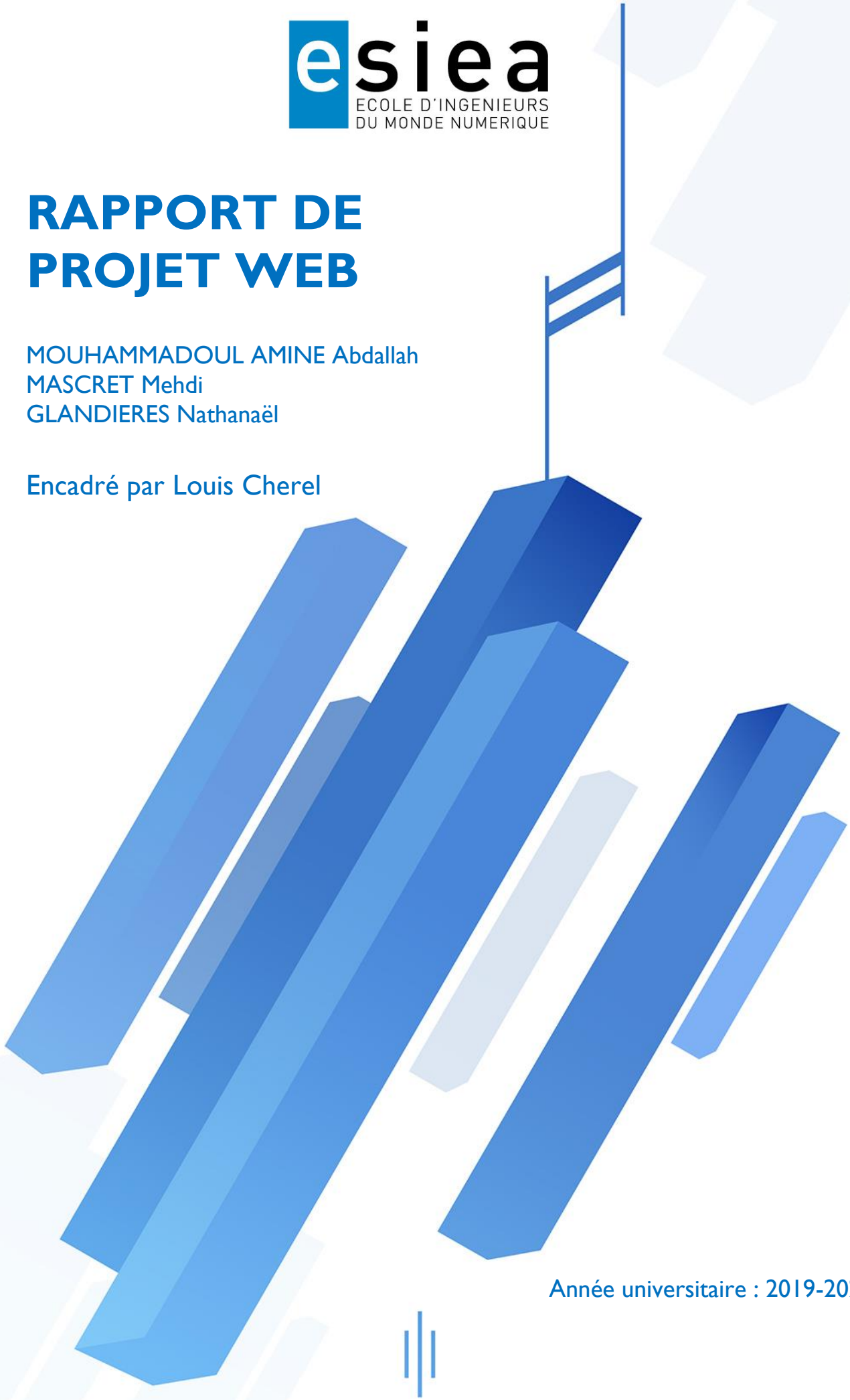
esiea
ECOLE D'INGENIEURS
DU MONDE NUMERIQUE

RAPPORT DE PROJET WEB

MOUHAMMADOUL AMINE Abdallah
MASCRET Mehdi
GLANDIERES Nathanaël

Encadré par Louis Cherel



Année universitaire : 2019-2020

1 Description du projet et réponse aux contraintes

Ce projet consiste à afficher un tableau récapitulant les items du jeu Minecraft. Les items sont répartis par thème (basiques, défense, nourriture et outils). Des onglets permettent d'accéder aux différents thèmes. Pour chaque item, les informations suivantes sont fournies :

- Le nom de l'item
- Les ingrédients nécessaires pour crafter l'item
- Une image pour indiquer comment disposer les ingrédients
- Une description de l'item, à quoi il sert

Le projet a été réalisé en utilisant HTML, CSS et javascript avec le framework VueJS côté client. Côté serveur, nous utilisons nodeJS avec son framework Express.

En plus des technologies mentionnées, ci-dessous les contraintes qui nous étaient exigées et comment ce projet répond à chacun d'eux :

- Authentification d'utilisateur ➔ Le site sert un peu de doc pour les items donc un utilisateur lambda n'a pas besoin de s'identifier pour accéder au contenu. Cependant, nous avons créé quelques utilisateurs admins. Quand ceux-ci se connectent, ils peuvent modifier les données dans les tableaux. Nous utilisons les cookies et sessions pour les authentifier après vérification de leurs identifiants.
- Intéractions CRUD avec le serveur ➔
    - Quand un admin est connecté, il peut créer un item en remplissant la dernière ligne du tableau et en appuyant sur le bouton + à droite. La liste d'item est alors directement modifée sur le serveur → Create OK
    - Les items se situent sur le serveur, le client les récupère grâce à une requête GET avec Axios. → Retrieve OK
    - Quand un admin est connecté, il a accès à un bouton à droite de chaque item qui lui permet de passer en mode « edit » et là il peut modifier soit le nom de l'item, ses ingrédients, son image et/ou sa description. Il a ensuite un bouton pour valider le changement et ainsi sortir du mode « edit ». La donnée est modifiée sur le serveur → Update OK
    - Quand un admin est connecté, il a accès à un bouton à droite de chaque élément, lui permettant de supprimer ce dernier. La suppression est effectuée directement sur le serveur → Delete OK
- Persistance des données, optionnel ➔ Le serveur récupère les données (items) dans des fichiers JSON dès qu'il est lancé. Les modifications mentionnées ci-dessus (create, update, delete) ne sont en réalité pas persistées automatiquement, donc si le serveur redémarre, elles disparaissent. C'est fait pour ne pas écrire trop souvent sur le filesystem (lent) mais aussi car l'admin peut, tant qu'il n'a pas cliqué sur le bouton « save », annuler les modifications en cliquant sur le bouton « cancel ». Ces boutons sont accessibles dans l'espace admin (menu haut gauche). Par contre, dès qu'il appuie sur le bouton « save », les données sont écrites sur le filesystem et donc persistées.
- Le projet a été déployé sur Glitch

2 Conception et Réalisation

2.1 Back-End

Niveau back end, nous avons tout d'abord créé une route permettant de récupérer les items grâce à une requête GET. C'est la plus simple car il n'y a pas besoin d'être connecté pour voir les items. Les données sont lues grâce à FS et la méthode readFileSync. Nous avons ensuite créé une route pour se login. On y vérifie si les identifiants envoyés dans le body de la requête sont bons et si c'est le cas, on attribue une ID à l'utilisateur ainsi qu'un statut admin à True.

Suite à cela, nous avons créé la route qui crée un item grâce à une requête POST et les données de l'item envoyées dans le body. A partir de là, nous avons créé un middleware où on vérifie si le statut admin est défini. Si oui, on continue, sinon on termine la requête en renvoyant une erreur au client lui indiquant qu'il doit se connecter. On applique le middleware pour toutes les requêtes nécessitant d'être admin.

Une fois que le post fonctionnait bien, nous avons créé une route pour update et une autre pour delete. Ils consistent tout simplement à retrouver l'indice de l'élément à supprimer ou mettre à jour et de le faire.

Pour finir, nous avons créé : une route pour se déconnecter (en supprimant la session et le cookie correspondant), une route pour annuler les modifications non persistées (en utilisant un tableauOriginal et un tableauCopy pour les données et en restaurant le tableauOriginal en cas d'annulation) et une route pour sauvegarder les modifications en les persistant.

Afin de tester nos routes, nous avons utilisé le logiciel Postman car c'est plus rapide que de faire un code client, et s'il y a une erreur, on sait que ça vient de l'api et non de notre code.

2.2 Interactions clients serveur

Toutes les interactions entre le client et le serveur sont faites dans le store Vuex. Les requêtes sont émises grâce à Axios. Nous utilisons des « actions » pour effectuer des opérations asynchrones car dans les outils de développements, l'opération est affichée seulement à partir du moment où on arrive dans le then et donc qu'on commit. Dès que les données sont récupérées, nous utilisons commit pour appeler un mutateur et c'est ce dernier qui met à jour les données dans le state. Nous créons alors des getters pour récupérer les variables du state ailleurs. Pour communiquer avec le serveur, nous avons donc simplement à déclencher l'action correspondante et récupérer la variable souhaitée grâce à un getter.

2.3 Front-End

VueJS nous offre un gros avantage, celui de pouvoir créer des composants. Un composant correspond à un fichier .vue et regroupe son template (html/css) et sa logique (javascript). Cela permet de facilement diviser les parties du site, chacune ne dépendant que peu ou pas des autres. Ça permet aussi de pouvoir réutiliser un composant. En particulier dans un projet de groupe, c'est très pratique de pouvoir se diviser les tâches : chacun fait un (ou plusieurs) composant(s) pour réaliser sa tâche. Vuex, que nous avons aussi utilisé, permet d'avoir toutes les données du site à un seul endroit, le store. Cela simplifie le débogage grâce aux outils de développement qui nous

indiquent l'état de chaque variable après chaque action. Vue router aussi est très utile, le site étant une single page application, il permet de savoir quel composant afficher en fonction de ce que rentre l'utilisateur dans la barre de recherche.

Nous avons donc commencé par créer un composant général permettant d'afficher un tableau avec les items et éventuellement les boutons d'admin en fonction de si l'utilisateur est connecté ou non. Ce composant prend en paramètres (props) un tableau d'items. Ensuite, pour chaque thème, nous avons créé un composant. Ce dernier appelle tout simplement le composant général et lui passe en paramètres les données correspondantes. Les données sont récupérées depuis le store grâce à des getters.

Nous avons ensuite créé un composant correspondant à barre de navigation. Celle-ci est composé des quatre onglets pour accéder aux différents thèmes ainsi qu'un bouton pour se connecter. Nous avons ajouté un listener pour chaque bouton, qui à chaque clic, charge le composant correspondant grâce à vue router.

Puis, nous avons créé une fenêtre pop-up contenant un formulaire pour pouvoir se connecter en renseignant son nom et mot de passe.

Enfin, nous avons créé un composant correspondant à un espace admin. Celui-ci apparait quand l'utilisateur se connecte et clique sur le menu en haut à gauche. Il peut y voir son image de profil ainsi que trois boutons pour valider ses modifications, les annuler ou se déconnecter.

3 Déploiement

Pour déployer, nous avons tout d'abord enlever toute référence à l'ip du serveur dans nos fichiers .vue. Nous faisons cela car au départ, nous avons deux serveurs, un pour servir le front-end du site et le deuxième pour servir les données (api). Or ce que nous devions faire avant de déployer, c'est d'avoir un seul serveur pour tout. Lorsque nous passons par la route /api, on accède aux données, sinon, on accède au html. Pour cela, nous avons fait un npm run build, ce qui a pour effet de rassembler tous les fichiers vue en un bundle avec un fichier html et 3 fichiers javascript. C'est à ce bundle que nous associons la route racine.

Ensuite, nous avons rajouté, dans le fichier package.json, un script start qui permet de lancer le serveur.

Finalement, nous avons tout push sur github, et une fois rendu sur le site glitch.com, nous avons tout simplement à cloner le projet depuis github. Notre site est déployé et nous obtenons un lien pour y accéder.

Lien du site : https://minecraft-vuejs.glitch.me
Identifiants :

- Name : zoro
- Password : sabre

Ou

- Name : luffy
- Password : viande

4 Difficultés rencontrées

- Fixer la barre de navigation (pour qu'elle reste en haut quand on scroll), les solutions sur le site ne marchaient pas (ajouter la propriété 'fixed'). On a fini par trouver que que la v-toolbar n'est plus fixable et qu'il faut utiliser v-app-bar
- Mettre le bouton de connexion à droite sur la barre de navigation. Il fallait utiliser un <v-spacer>
- Mettre un background à la page : seul les inline styles fonctionnent
- Beaucoup de temps pour se documenter (vuejs, vuex, vuetify, express, express-sessions)