Hugo Dalboussière
Arthur Catrisse

LE MAUVAIS COIN

Description du projet

Le mauvais coin est une application web d'achat/vente inspirée de Leboncoin. Les utilisateurs peuvent consulter les annonces mises en ligne et contacter les vendeurs. Si un utilisateur crée son compte, il peut alors publier et administrer ses propres annonces.

Liens

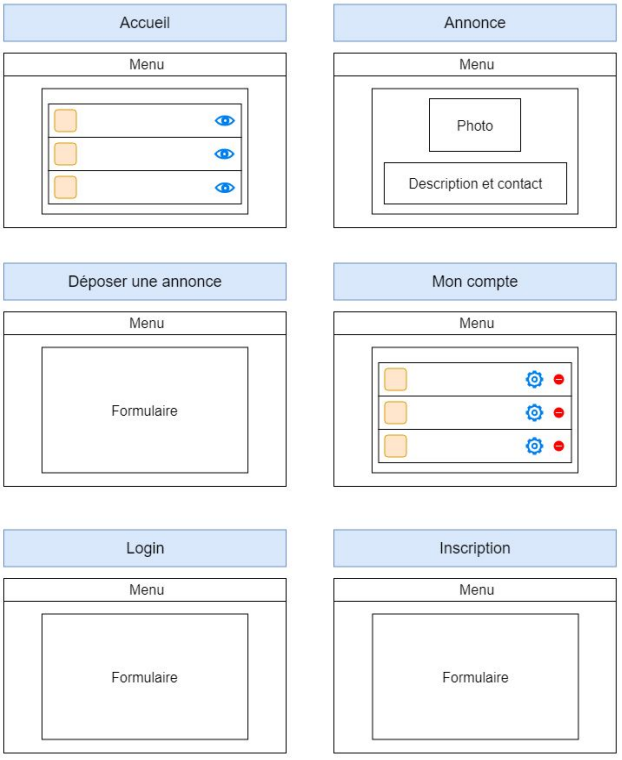
https://github.com/turgodi/lemauvaiscoin
https://le-mauvais-coin.herokuapp.com
(Attention, 1ère connexion lente car démarrage serveur)

Conception

Tout projet débute par une phase de conception. Nous nous sommes mis à la place de l'utilisateur final afin de définir les besoins fonctionnels de notre application :

- Consulter les annonces
- Se connecter
- Se déconnecter
- Administrer ses annonces (création, suppression)

Aussi, nous avons réalisé une maquette définissant les écrans utilisateurs nécessaires.



Réalisation

Back-end

Nous avons utilisé le langage Node.js pour le back-end.
Le back-end (côté serveur) est une API suivant l'architecture REST. Son rôle est de gérer toute la business logic de l'application, à savoir la gestion des annonces et des authentifications utilisateur.
L'avantage principal d'une API REST est le fait de pouvoir supporter plusieurs applications clientes utilisant un environnement différent (Ex. un site web et une application iOS).

Front-end

Le front-end (côté client) est un site web tirant parti de HTML/CSS mais surtout de Javascript et du framework Vue.js.
Le rôle du front-end est uniquement d'afficher le contenu de l'application. Vue.js nous permet de faire une Single Page Application (SPA) ce qui présente plusieurs avantages :

- Meilleures performances (communications légères en JSON, le serveur n'envoie le contenu HTML qu'une seule fois)
- Sensation de fluidité lors de la navigation (le contenu HTML est chargé depuis le cache du client)
- Séparation distincte avec le back-end pour une meilleure maintenabilité
- Mise en cache de l'application dans le navigateur après la première visite

Versioning

Nous avons utilisé Git et GitHub pour le versioning de l'application et le travail collaboratif. GitExtensions a été utilisé comme logiciel pour visualiser de manière plus intuitive notre projet sur GitHub.
Au total, une petite cinquantaine de commits ont été envoyés pendant toute la durée du projet.
Nous avons aussi testé des outils d'intégration continue (CircleCI) qui s'intègrent nativement à GitHub. Cela permet d'avoir une idée d'une gestion de projet plus réaliste avec des tests unitaires qui sont exécutés automatiquement lors de la création d'une pull request. Si les test passent, on peut merger la pull request, sinon elle est refusée.

Déploiement

Nous avons utilisé Heroku pour l'hébergement de notre application.
C'est un service permettant de déployer rapidement une application et de la monitorer.
Le service se connecte directement à un repo GitHub comme avec Glitch, ce qui permet d'accélérer le processus de déploiement. Notre application est accessible à l'URL suivante : https://le-mauvais-coin.herokuapp.com/. Attention, comme nous utilisons la version

gratuite de Heroku, le serveur se met en veille tout seul et met environ trente secondes à se relancer tout seul lors de la première connexion depuis la dernière mise en veille.

Difficultés rencontrées

Front-end

Nous n'avions jamais utilisé Vue.js précédemment ce qui était un premier challenge. Heureusement, cela reste du Javascript et la documentation officielle est bien fournie. De plus, une grande communauté est présente pour trouver rapidement des solutions aux problèmes rencontrés sur des forums. Le temps d'adaptation n'était pas si long.

Nous avons eu des difficultés à utiliser les variables d'environnement avec les fichiers .env car il y avait apparemment beaucoup de manières différentes de faire mais nous avons finalement utilisé les conventions de nommage ".env.nom-environnement".
Ainsi en environnement de développement, l'application utilisera les variables d'environnement du fichier ".env.development".

Une autre difficulté a été de faire communiquer de l'information entre différents components. Un cas spécifique : l'utilisateur se connecte avec son identifiant et son mot de passe. On souhaite le rediriger automatiquement à la page home principale si la connexion a bien réussi. Malheureusement, la page principale ne reconnaît pas que l'utilisateur est connecté. Afin de résoudre ce souci, nous avons instancié un "bus", un nouvel objet Vue pour communiquer entre différents components. Dans la page Vue principale, on ajoute un "watch" qui envoie un appel de fonction particulier au bon component au moment où la route souhaitée en empruntée (ici entre le formulaire de login et la page home).

Une autre difficulté était d'utiliser une architecture de l'application relativement propre. C'est difficile lorsque l'on ne connaît pas le framework. Nous avons créé un dossier components qui contenait tous nos composants (9 au total) et un dossier services qui contenait des classes statiques permettant de faire les appels REST au back-end.

Back-end

Node.js n'était pas familier non plus, ainsi il fallait bien comprendre que c'était du Javascript exécuté côté serveur.

Nous n'avons pas réussi à utiliser des classes pour travailler réellement en orienté objet comme nous l'aurions souhaité. ainsi nous avons dû exporter des fonctions statiques dans le module directement avec des "exports.fonction = …".

L'utilisation des Json Web Token avec la librairie node-jsonwebtoken pour l'authentification était intéressante mais nous avions du mal à assimiler le fonctionnement au début. Après avoir compris, nous avons pu mettre en place des solutions élégantes pour sécuriser les

endpoints de l'api REST comme un middleware "isLoggedIn" qui vérifie la présence et la validité d'un token dans la requête, puis renvoie une erreur ou passe la main à la fonction suivante.

Enfin, une gestion propre des cas d'erreurs a suscité beaucoup de réflexion. Nous avons finalement décidé que les contrôleurs ne serviraient que de point d'entrée et de sortie des requêtes, mais que la logique métier serait contenue dans les services. Si une erreur apparaît dans un service, le contrôleur "attrape" (catch) l'erreur et renvoie une réponse d'erreur appropriée au client, par exemple un code HTTP 400 avec un message "Invalid parameters" si le client n'envoie pas les bonnes données attendues.

Versioning

Nous avons de l'expérience avec Git pour des projets basiques mais avons encore eu des soucis lorsqu'il fallait résoudre des conflits. Nous avons utilisé Git Extensions pour avoir une interface visuelle et KDiff pour résoudre les conflits mais cela ne nous a pas facilité la tâche: le faire manuellement directement dans le fichier était plus facile. C'est sans doute dû à un manque d'expérience avec les logiciels.

Nous avons voulu essayer des outils d'intégration continue comme CircleCI pour aller plus loin, ainsi nous avons dû lire la documentation de l'outil pour l'installer et le lier à notre repo GitHub. Ainsi, les commits sur master étaient bloqués et il fallait obligatoirement faire des pull requests. Une fois la pull request envoyée, cela notifiait CircleCI qui lançait les tests (factices, nous n'avons pas eu le temps de faire de réels tests unitaires). Si les tests passaient, alors le bouton "Merge Pull Request" devenait vert sur GitHub et on pouvait confirmer le merge dans master de la branche.
Nous n'avons pas utilisé ce fonctionnement tout au long du projet, mais c'était intéressant de le mettre en place pour simuler le développement d'un produit avec une plus grande équipe dans un cadre professionnel.

Déploiement

Le déploiement nous a posé beaucoup de problèmes. Jusqu'à la fin, nous travaillons en local et tout marchait bien. Les serveurs de développement back-end et front-end avaient leurs propres ports.
Nous avons alors essayé de déployer sur Glitch mais pour une raison qui nous échappe nous n'avons pas trouvé l'option de déploiement d'application Node.js. Ainsi nous avons utilisé Heroku, un concurrent qui propose aussi une option gratuite.
Le problème d'Heroku est qu'il ne supporte pas une application qui utilise deux ports différents, et peu importe le port utilisé il sera remappé par Heroku sur le port 80.
Ainsi à la fois l'hébergement du front-end et l'api REST du back-end tournent sur le même port… Ce n'est pas idéal et nous avons dû faire des modifications en conséquence sur une branche Git dédiée à Heroku mais le résultat est fonctionnel.