Documentation Technique

Code Frontend - BlogAura

Projet DEV Learn IT B3

Février 2026

Analyse du code source React

Introduction

Ce document présente une analyse complète du code frontend de BlogAura, notre projet de blog développé en React. J'ai essayé de détailler tous les composants et la logique derrière chaque fonctionnalité.

Le projet utilise React 19 avec Vite 7 comme bundler et Tailwind CSS 4 pour le styling. L'architecture suit un pattern assez classique avec des contextes pour l'état global et une organisation en composants réutilisables.

1. Architecture du projet

1.1 Structure des dossiers

Le projet est organisé de façon assez standard pour une app React. Voici comment j'ai structuré les fichiers :

Dossier/Fichier	Rôle	Nombre de lignes
src/main.jsx	Point d'entrée de l'app	33
src/App.jsx	Composant principal + routes	132
src/contexts/	Gestion état global (Auth, Theme)	167
src/components/	Composants réutilisables	633
src/pages/	Pages de l'application	622
src/data/	Données de test	183

Au total ça fait environ 1800 lignes de code, ce qui est pas énorme mais suffisant pour un projet étudiant avec toutes les fonctionnalités qu'on voulait.

2. Point d'entrée et initialisation

2.1 Le fichier main.jsx

C'est le fichier qui lance toute l'application. Il utilise createRoot de React 18+ pour monter l'app dans le DOM. Ce qui est intéressant c'est l'ordre des Providers qu'on imbrique les uns dans les autres (comme des poupées russes) :

```
StrictMode → BrowserRouter → ThemeProvider → AuthProvider → App
```

J'ai mis dans cet ordre parce que le Router doit englober tout ce qui utilise la navigation, le ThemeProvider est indépendant de l'auth, et l'AuthProvider doit être accessible partout.

2.2 Le composant App.jsx

C'est vraiment le cœur du projet. App.jsx gère tout l'état des posts et des commentaires avec useState. Au début j'avais pensé utiliser Redux mais finalement c'était trop compliqué pour ce qu'on faisait, donc j'ai gardé du state local qui est passé en props.

Les trois fonctions principales qu'on y trouve :

• **addPost(newPost)** : Ajoute un nouveau post au début du tableau. J'utilise Date.now() pour générer l'ID, c'est pas parfait mais ça marche.

• **addComment(postId, comment)** : Similaire mais pour les commentaires. Le commentaire est lié au post via le postId.

• **toggleReaction(postId, emoji, userId)** : C'est la plus compliquée. Elle gère l'ajout/retrait des réactions emoji. Si l'user a déjà réagi, on retire sa réaction, sinon on l'ajoute. Si un emoji n'a plus de réactions, on le supprime complètement.

3. Les contextes React

3.1 AuthContext

Ce contexte gère toute l'authentification. Bon, dans notre cas c'est une fausse auth parce qu'on n'a pas de vrai backend, mais ça simule le comportement. Les infos utilisateur sont stockées dans le localStorage pour persister entre les sessions.

Les fonctions principales :

• login(email, password) : Vérifie les credentials et stocke l'user

• logout() : Supprime l'user du localStorage

• Le hook useAuth() pour accéder facilement au contexte

Pour l'instant on a trois users de test : un user normal, un admin et un auteur. Les mots de passe sont en dur dans le code, ce qui serait évidemment pas acceptable en prod, mais pour le projet ça suffit.

3.2 ThemeContext

Gère le dark mode / light mode. Au chargement, il vérifie dans le localStorage s'il y a une préférence sauvegardée, sinon il utilise la préférence système du navigateur avec window.matchMedia('(prefers-color-scheme: dark)').

Quand le thème change, on ajoute ou retire la classe 'dark' sur le documentElement, et Tailwind CSS fait le reste avec ses classes dark:*.

4. Composants principaux

4.1 Navbar

La barre de navigation en haut. Elle affiche le logo, les liens de navigation, le bouton de toggle du thème, et les infos de l'utilisateur connecté. Si l'user est admin, il voit en plus le bouton 'Nouveau Post'.

4.2 PostCard

Le composant le plus important je dirais. Il affiche une carte pour chaque article avec le titre, l'auteur, la date, les tags, et un extrait du contenu. Il y a deux modes d'affichage : compact (pour la page d'accueil) et détaillé (pour la page article).

Ce qui est cool c'est qu'on peut cliquer sur 'Lire la suite' et ça ouvre une modale avec le contenu complet, sans changer de page. J'ai utilisé le composant Modal pour ça.

4.3 EmojiReactions

Affiche les boutons de réaction avec les emojis. Chaque emoji a un compteur qui montre combien de personnes ont réagi. Si l'user a déjà réagi avec un emoji, le bouton est mis en surbrillance (background

bleu). C'est géré par la fonction toggleReaction dans App.jsx.

4.4 CommentSection

Gère l'affichage des commentaires et le formulaire pour en ajouter. Les commentaires sont paginés (5 par page) pour pas que ça soit trop long. J'ai ajouté des boutons Précédent/Suivant pour naviguer entre les pages.

Un truc que j'ai galéré dessus c'est le formatage des dates. J'utilise toLocaleDateString avec la locale 'fr-FR' pour avoir les dates en français, mais au début ça marchait pas bien avec les fuseaux horaires.

4.5 Modal

Une fenêtre modale réutilisable. Elle utilise useEffect pour bloquer le scroll du body quand elle est ouverte, et pour écouter la touche Échap pour fermer. Le cleanup dans le return du useEffect remet tout en place quand la modale se ferme.

5. Les pages

5.1 Home

La page d'accueil affiche tous les posts en grille. Il y a un système de filtrage par tags en haut : on peut cliquer sur un tag pour voir uniquement les posts qui ont ce tag. C'est géré avec un state selectedTag qui filtre le tableau de posts.

5.2 Login

Formulaire de connexion classique avec email et password. Quand on submit, ça appelle la fonction login du AuthContext. Si les credentials sont bons, on est redirigé vers l'accueil avec useNavigate. Sinon on affiche un message d'erreur.

J'ai ajouté des indications pour les comptes de test parce que sinon personne saurait comment se connecter vu qu'on a pas de page d'inscription.

5.3 CreatePost

Formulaire pour créer un nouveau post. Accessible uniquement aux admins grâce au composant ProtectedRoute. On peut choisir le titre, le contenu, et sélectionner plusieurs tags dans une liste.

Les tags sélectionnés sont gérés dans un tableau avec useState. Quand on clique sur un tag, soit on l'ajoute soit on le retire avec une logique toggle.

5.4 PostDetail

Page dédiée pour un article complet. On récupère l'ID du post dans l'URL avec useParams, puis on trouve le post correspondant dans le tableau. Si le post existe pas, on affiche 'Post non trouvé'.

Cette page affiche le post complet avec les réactions et les commentaires. C'est presque la même chose que dans la modale, mais en version standalone.

5.5 About

Page 'À propos' qui présente l'auteur du blog (fictif). Elle affiche une photo, une bio, des stats (nombre de posts, de followers), et des liens vers les réseaux sociaux. C'est surtout pour montrer qu'on sait faire des mises en page sympas avec Tailwind.

6. Gestion de l'état et flux de données

Toute la gestion de l'état se fait avec les hooks useState et useContext. Pas besoin de Redux ou autre librairie externe pour ce projet.

Les données principales (posts et comments) sont dans App.jsx

Les données d'auth sont dans AuthContext

Le thème est dans ThemeContext

Le pattern utilisé c'est le 'lifting state up' : l'état est stocké en haut de l'arbre des composants et les fonctions de modification sont passées en props vers le bas. Par exemple, quand on ajoute un commentaire, le flux est :

```
CommentSection (user tape) → onAddComment prop → PostCard (construit l'objet) →
App.addComment (modifie le state) → React re-render
```

7. Les hooks React utilisés

7.1 useState

Le hook le plus utilisé dans le projet. Permet de gérer l'état local des composants. Par exemple dans Login on a des states pour email, password et error.

7.2 useEffect

Utilisé pour les effets de bord. Par exemple dans AuthContext on charge les infos utilisateur depuis le localStorage au démarrage avec un useEffect qui a un tableau de dépendances vide [].

Dans Modal, le useEffect écoute la touche Échap et bloque le scroll. La fonction return fait le cleanup quand le composant se démonte.

7.3 useContext

Pour accéder aux contextes depuis n'importe quel composant. On a créé des hooks custom useAuth() et useTheme() qui encapsulent useContext, c'est plus propre que d'importer useContext à chaque fois.

7.4 useNavigate

De React Router. Permet de faire de la navigation programmatique. Par exemple après un login réussi on fait navigate('/') pour rediriger vers l'accueil.

7.5 useParams

Aussi de React Router. Récupère les paramètres de l'URL. Dans PostDetail on l'utilise pour récupérer l'ID du post à afficher.

8. Styling avec Tailwind CSS

J'ai choisi Tailwind pour le styling parce que c'est rapide et ça évite d'écrire du CSS. La configuration est dans tailwind.config.js avec les couleurs custom qu'on utilise.

Pour le dark mode, on utilise les classes dark:* de Tailwind. Par exemple : 'bg-white dark:bg-gray-800' affiche un fond blanc en mode clair et gris foncé en mode sombre.

J'ai aussi ajouté un plugin @tailwindcss/line-clamp pour tronquer les textes longs. Ça permet de faire line-clamp-3 pour afficher uniquement 3 lignes avec des ... à la fin.

9. Données de test (mockData.js)

Comme on a pas de vraie base de données, j'ai créé un fichier mockData.js qui contient toutes les données de test : les posts, les commentaires, les emojis disponibles, les tags, et les infos sur l'auteur.

Les posts ont une structure avec id, title, content, author, date, tags, et reactions. Les reactions sont un objet où chaque emoji est une clé avec un tableau d'userIds.

Exemple de structure : reactions: { '■': [1, 3], '❤■': [2] } signifie que les users 1 et 3 ont mis un pouce et l'user 2 un cœur.

10. Bonnes pratiques appliquées

10.1 Immutabilité du state

C'est super important en React de jamais modifier le state directement. On crée toujours de nouvelles copies avec le spread operator (...).

Par exemple pour toggleReaction, au lieu de faire posts[0].reactions['■'].push(userId), on fait une copie complète : {...post, reactions: {...post.reactions, ...}}

10.2 Composants contrôlés

Tous les inputs sont contrôlés par React. La valeur vient du state et onChange met à jour le state. Comme ça React a toujours le contrôle sur les données.

10.3 Séparation des responsabilités

Chaque composant a une seule responsabilité. Par exemple PostCard s'occupe de l'affichage, mais c'est CommentSection qui gère les commentaires. On aurait pu tout mettre dans PostCard mais ça aurait été le bordel.

11. Difficultés et apprentissages

Le projet m'a pris pas mal de temps mais j'ai appris plein de trucs. Voici les principales difficultés que j'ai rencontrées :

Les réactions emoji : Au début je stockais juste un compteur par emoji, mais du coup je pouvais pas savoir si l'user avait déjà réagi ou non. J'ai dû refaire pour stocker un tableau d'userIds, et ça a compliqué la logique.

Le dark mode : J'avais pas bien compris au début comment Tailwind gère le dark mode. Il faut ajouter la classe 'dark' sur le html et ensuite tous les dark:* fonctionnent. Une fois que j'ai pigé c'était facile.

La pagination des commentaires : Calculer quelle page afficher et gérer les boutons Précédent/Suivant, ça m'a pris un moment à bien faire.

Les routes protégées : Comprendre comment protéger certaines routes avec ProtectedRoute et vérifier si l'user est admin. Au début ma condition était pas bonne et ça marchait pas.

12. Améliorations possibles

Si j'avais plus de temps ou pour une vraie prod, voici ce que je ferais :

• Connecter à un vrai backend avec une API REST ou GraphQL

• Gérer la persistance des données (pour l'instant tout est perdu au refresh)

• Ajouter des tests unitaires avec Jest et React Testing Library

• Optimiser les performances avec React.memo et useMemo

• Ajouter un système de recherche pour filtrer les posts

• Gérer le responsive encore mieux pour mobile

• Ajouter des animations et transitions plus smooth

Conclusion

Ce projet m'a permis de bien approfondir React et de comprendre comment structurer une application frontend moderne. L'utilisation des contextes pour l'état global, les hooks pour la logique métier, et Tailwind pour le styling forment un stack vraiment efficace.

La partie la plus intéressante pour moi c'était de gérer le système de réactions et les commentaires. Ça demande de bien comprendre l'immutabilité et la gestion d'état en React.

Globalement je suis content du résultat, même si y'a encore des trucs à améliorer. C'est un bon projet de portfolio qui montre qu'on peut créer une app complète avec React.

Document rédigé en février 2026

Projet BlogAura - DEV Learn IT B3