ChatGPT

Documentation de l'application HNR Monitor (Suivi des habitats non réglementaires)

Introduction générale

L'application HNR Monitor est un outil web interne destiné aux agents d'autorité, aux fonctionnaires et aux superviseurs hiérarchiques pour le suivi des habitats non réglementaires. Son objectif principal est de faciliter l'administration et le suivi des opérations sur le terrain : gestion des **missions** de surveillance, signalement des **changements** détectés (extensions illégales, nouvelles constructions), enregistrement des **actions** menées (démolitions, signalements, régularisations), et génération de **procès-verbaux (PV)** correspondants. L'interface propose un **tableau de bord** synthétique, une **carte interactive** permettant de visualiser géographiquement les points d'intérêt, une gestion des **notifications** en temps réel, et des fonctionnalités d'administration comme la gestion des utilisateurs.

En résumé, l'application implémente une logique de tableau de bord pour le suivi global, des pages dédiées à chaque entité métier (missions, changements, actions, utilisateurs), et une navigation protégée par authentification et permissions. Elle vise à centraliser toutes les informations liées aux habitats non réglementaires : localisation des foyers (`douars`), changements détectés par satellite ou drone, missions planifiées ou en cours, actions d'intervention effectuées par les agents, et rapports officiels (PV) générés. Le tout est conçu de manière **responsive** (adapté mobile) et accorde de l'importance à l'**accessibilité** (navigation clavier sur la carte, libellés ARIA, etc.).

Structure du projet

Le code source est organisé en plusieurs dossiers correspondant chacun à un aspect de l'application :

- `components/` – Composants d'interface réutilisables. On y trouve des composants UI comme `ProtectedRoute` (gestion de l'accès aux routes protégées), des composants liés à la carte (`CarteFilters` , `CarteListe` , `CartePanel` pour les filtres, la liste des points et le panneau de détails sur la carte), des composants d'upload de fichiers (`FileUpload/*` pour charger des photos/documents), la mise en page principale (`Layout/*` dont `MainLayout` pour le menu et l'en-tête), des modales spécifiques (ex: `MissionCreationModal`), le menu de navigation (`Navigation`), et le dropdown de notifications (`Notifications`). Chaque sous-dossier ou fichier correspond à un composant React isolé avec parfois son fichier de style module CSS associé.

- `pages/` – Composants de page correspondant aux écrans/routes de l'application. Par exemple : `Dashboard` (tableau de bord), `Carte` (vue cartographique), `Missions` (liste des missions + création/nouveauté), `Actions` (liste des actions + formulaire de nouvelle action), `Changements` (liste des changements détectés + déclaration), `Utilisateurs` (gestion des comptes utilisateurs), `Login` (page de connexion), `NotFound` (404), etc. Ces composants de page orchestrent les composants plus petits et contiennent la logique spécifique à chaque vue (chargement des données, interactions utilisateur, etc.). Ils utilisent souvent les **contexts** et les **services** pour accéder aux données.

- `contexts/` – Contextes React pour le state global, incluant `AuthContext` (authentification, informations de l'utilisateur connecté, rôles, permissions) et `NotificationContext` (gestion centralisée des notifications temps réel). Ces contextes fournissent via React Context API des hooks (`useAuth`, `useNotifications`) utilisés à travers l'application pour connaître l'état global (ex: utilisateur courant, notifications non lues) et exécuter des actions globales (ex: login, logout, envoi d'une notification).

- `services/` – Couche d'accès aux données et simulations d'API. On y trouve `apiProvider.js` (prévu pour centraliser les appels API réels), et surtout `mockApi.js` et `mockData.js` qui fournissent une API simulée côté front-end. `mockData.js` contient des données statiques simulant la base de données (utilisateurs, missions, douars, changements, actions, fichiers, PV, etc.), tandis que `mockApi.js` expose une classe `MockApiService` avec des méthodes CRUD asynchrones pour manipuler ces données en mémoire (ex: `getMissions()`, `createAction()`, `login()`...). Il inclut aussi des mécanismes comme un délai simulé et un système de **subscribe/notify** pour mettre à jour l'UI lors des modifications de données. Enfin, un hook React `useMock()` est fourni pour récupérer facilement les données mock (missions, changements, actions, etc.) et réagir aux mises à jour.

- `utils/` – Fonctions utilitaires et helpers pour des tâches transverses. Par exemple `actionHelpers.js` (fonctions liées aux actions et PV : types d'actions disponibles, génération d'identifiant d'action et de contenu automatique de PV, etc.), `fileHelpers.js` (validation des fichiers avant/après, etc.), `geolocationHelpers.js` (gestion de la géolocalisation navigateur, formatage de coordonnées), `notificationHelpers.js` (potentiellement pour formatage de notifications – à implémenter), et un `index.js` d'export. Ces utilitaires factorisent des logiques complexes hors des composants, par exemple la génération d'un PV automatique via `createAutoPV()` qui prépare un brouillon d'après une action.

- `styles/` – Feuilles de style globales ou partagées. L'application utilise principalement la bibliothèque Ant Design (antd) qui apporte ses styles, mais ce dossier contient des ajustements personnalisés : `components.css` définit des variables CSS globales (couleurs principales, etc.) et quelques styles globaux pour l'app (surcharges ou ajouts spécifiques), tandis que `common.module.css` fournit des classes CSS modulaires réutilisables (par ex. styles pour des en-têtes de section, grilles, etc., que les composants peuvent importer sans collision de noms grâce aux modules CSS).

En plus de ces dossiers principaux, à la racine du `src` on trouve le point d'entrée `index.js` qui monte l'application React (et configure la locale française pour la librairie dayjs), ainsi que `App.js` qui déclare le routeur et englobe l'ensemble dans les contextes d'authentification et de notifications.

Description détaillée fichier par fichier

Point d'entrée et configuration

- `src/index.js` : Initialise l'application React en important `App` et en utilisant `createRoot` pour l'attacher au DOM. Ce fichier importe aussi `index.css` (styles globaux de base) et configure dayjs en français. C'est là que l'arborescence de composants est enveloppée, via `<App />`, dans la div racine du HTML.

- `src/App.js` : Composant principal qui définit le **routeur** et le contexte global. Il importe les context providers `AuthProvider` et `NotificationProvider` pour englober toute l'application. La configuration Ant Design est appliquée via `ConfigProvider` (locale française, thème couleur primaire, etc.).

La structure du routeur utilise `<BrowserRouter>` et déclare toutes les routes de l'application à l'aide de `<Routes>` et `<Route>`. Les pages principales sont chargées en lazy loading (code splitting) pour optimiser les performances, via `React.lazy` et `<Suspense>`.

Les routes sont organisées comme suit : - La route `/login` rend la page de connexion `Login` sans protection (publique). - Toutes les autres routes sont englobées dans une route parent "/" protégée qui rend le `MainLayout`. Ce layout inclut la navigation et un `<Outlet>` pour afficher le contenu des sous-routes. Ainsi, toute URL autre que `/login` nécessite d'être authentifié grâce au composant `ProtectedRoute` qui entoure `MainLayout`. - À l'intérieur de cette route protégée, on trouve les sous-routes : - `/dashboard` (tableau de bord) – page `Dashboard`. - `/carte` – page `Carte` (visualisation cartographique). - **Missions** : `/missions` liste des missions (`Missions`), `/missions/:id` détails d'une mission (`MissionDetails`), `/missions/nouveau` création d'une mission (`NouvelleMission`), et `/missions/selecteur-zone` sélection de zone (composant `SelecteurZone` pour définir une zone géographique sur carte). Les routes de création sont elles-mêmes protégées par des permissions fines : par ex. la création de mission exige la permission *canCreateMissions* et est donc enveloppée dans un `<ProtectedRoute requiredPermission="canCreateMissions">` (seul un **Membre DSI** a ce droit). - **Actions** : `/actions` liste des actions (`Actions`), `/actions/nouveau` formulaire de nouvelle action (`NouvelleAction`), `/actions/create` (alias éventuel vers `ActionCreate`), et `/actions/pv/:pvId` édition d'un PV (`PVEditor`). Là aussi, l'accès est protégé par *canCreateActions* pour les pages d'actions. *Note:* la permission utilisée est canCreateActions pour la liste, ce qui peut être un léger oubli – idéalement canViewActions suffirait pour permettre aux superviseurs (**Gouverneur**) de consulter la liste même s'ils ne peuvent en créer. - **Changements** : `/changements` liste des changements (`Changements`), `/changements/nouveau` déclaration d'un changement (`ChangementCreate`). Voirie protégée par *canViewChangements* (lecture pour tous les rôles) et *canCreateChangements* (création réservée au **Membre DSI**). - **Utilisateurs** : `/utilisateurs` gestion des utilisateurs (`Utilisateurs`), protégé par *canManageUsers* (seul le DSI peut gérer les comptes). - **Statistiques** : `/stats` ou `/statistiques` affichent la page `Stats`, protégé par *canViewStats* (DSI et Gouverneur). - Une route wildcard `*` rend la page `NotFound` pour toute URL non définie.

En somme, `App.js` définit le squelette de navigation de l'application. Il utilise abondamment `ProtectedRoute` (voir ci-dessous) pour conditionner l'accès en fonction de l'authentification, du rôle utilisateur ou de permissions précises.

Contexte d'authentification et route protégée

- `src/contexts/AuthContext.js` : Implémente le contexte d'authentification et de gestion des droits. Il exporte `AuthProvider` qui englobe l'UI et fournit les valeurs du contexte, ainsi que le hook `useAuth()` pour y accéder facilement.

Ce contexte maintient l'état de l'utilisateur connecté (`user`), un booléen `isAuthenticated` et un état de chargement `loading` (utile pendant la vérification initiale). Au montage, un effet `useEffect` vérifie s'il existe un token de session et un utilisateur stockés (via un utilitaire `storage`, probablement localStorage) et, le cas échéant, restaure ces infos pour maintenir la session. Si les données de session ne sont pas valides, on nettoie le stockage.

Il expose des fonctions majeures : - `login(email, password)` : tente une connexion via le service d'API mock (`mockApiService.login`) et, en cas de succès, stocke le token et l'utilisateur dans le stockage local puis met à jour le contexte (`user` et `isAuthenticated`). En cas d'échec, une notification d'erreur est affichée. On utilise les notifications d'Ant Design pour informer l'utilisateur du succès ou de l'erreur de connexion. - `logout()` : appelle l'API mock `logout` puis nettoie les données de session (retire token et user du storage) et réinitialise le contexte (user null, isAuthenticated false). Une notification d'information confirme la déconnexion. - `updateUser(newUserData)` : permet de mettre à jour le profil utilisateur en contexte et dans le storage (utile par ex. après modification du numéro de téléphone ou autres attributs).

Surtout, `AuthContext` définit la logique fine des **rôles et permissions**. Les rôles disponibles (synchronisés avec ceux du backend attendu) sont définis dans `USER_ROLES` (Agent d'Autorité, Membre DSI, Gouverneur). Le contexte dérive ensuite un ensemble complet de permissions booléennes via la fonction `hasPermission(permission)`. Par exemple : - *canCreateMissions* est true seulement si l'utilisateur est MEMBRE_DSI (seul le DSI peut créer une mission), - *canViewMissions* est true pour Gouverneur, DSI et agents (tout le monde peut voir les missions), - *canManageUsers* uniquement pour DSI, - etc. Il y a ainsi des permissions couvrant la création/suppression de missions, création/mise à jour d'actions, déclaration de changements, accès à la gestion des utilisateurs, vue des stats, export de données, etc. Ces permissions sont exposées via `contextValue.PERMISSIONS` et aussi par des méthodes utilitaires (ex: `isAgent()`, `isDSI()`, `isGouverneur()` pour tester le rôle courant, ou directement `canAccessUsers()` qui utilise hasPermission).

Le contexte fournit également une fonction `isReadOnly(resource)` qui renvoie true si l'utilisateur courant ne peut qu'en lecture seule sur une ressource donnée. La règle implémentée est que les agents d'autorité **peuvent modifier** uniquement les ressources (actions, missions) **de leur propre commune ou qu'ils ont créées**; sinon, ils sont en lecture seule. Les autres rôles (DSI, Gouverneur) ne sont jamais en read-only (ils peuvent tout modifier par ailleurs ou n'en ont pas besoin).

Enfin, `AuthContext` formate le nom du rôle (`getRoleDisplayName`) pour affichage (par ex. *MEMBRE_DSI* → "Membre DSI"), et expose `getUserLocation()` pour obtenir la commune/préfecture de l'utilisateur connecté facilement.

En résumé, ce fichier centralise l'**authentification** et le **contrôle d'accès**. Toutes les pages ou composants qui ont besoin de connaître l'utilisateur courant ou de vérifier un droit utilisent le hook `useAuth()`.

- `src/components/ProtectedRoute.js` : Un composant haut niveau qui protège l'accès à certaines routes selon l'état d'authentification ou les permissions. Il s'utilise en enveloppant une page dans le routeur (comme on l'a vu dans `App.js`). Son fonctionnement :
- Il utilise `useAuth()` pour récupérer `user`, `loading`, `isAuthenticated` et `hasPermission`.
- Si l'état d'auth est en cours de chargement (`loading` true, par ex. vérification du token au démarrage), il retourne un indicateur de chargement pleine page (un spinner avec texte "Vérification des accès...") pour patienter.
- Si l'utilisateur n'est pas authentifié (`!isAuthenticated`), il redirige vers la page de login, en passant dans l'état de navigation l'URL cible initialement demandée (pour un retour post-login).
- Si une prop `allowedRoles` est fournie et que le rôle de l'utilisateur n'est pas dans la liste, alors on affiche un message d'erreur 403. Ce message (« Rôle insuffisant ») est présenté via un composant visuel `<Result>` d'AntD dans une carte, indiquant à l'utilisateur que son rôle actuel

(formaté lisiblement) ne permet pas d'accéder à cette section. Il liste aussi quels rôles seraient autorisés, et propose des boutons pour revenir en arrière ou aller à l'accueil.
- Si une prop `requiredPermission` est fournie et que `hasPermission(requiredPermission)` retourne false, on affiche un autre message 403 (« Accès refusé ») détaillant la permission requise et le rôle actuel de l'utilisateur. Là encore, des boutons proposent de revenir ou d'aller au tableau de bord.
- Si toutes les conditions d'accès sont remplies, le composant rend simplement ses `children` (le contenu protégé).

Ainsi, `ProtectedRoute` permet de sécuriser non seulement l'accès global (auth ou non) mais aussi de filtrer par rôle ou permission spécifique. Il factorise le comportement d'affichage des erreurs d'accès en fournissant une UI cohérente en cas d'autorisation manquante. À noter qu'il formate les noms de rôle via une fonction interne `formatRole` cohérente avec `AuthContext`.

Exemple d'usage: dans `App.js`, la route `/missions/nouveau` est déclarée comme : `<ProtectedRoute requiredPermission="canCreateMissions"><NouvelleMission /></ProtectedRoute>`. Concrètement, cela signifie que si un agent sans droits tente d'accéder à `/missions/nouveau`, il verra le message d'erreur "Rôle insuffisant" ou "Accès refusé" selon son cas, au lieu du formulaire de création.

Contexte des notifications et menu déroulant

- `src/contexts/NotificationContext.js` : Gère l'état global des notifications dans l'application. Ce contexte utilise un `useReducer` pour manipuler une liste de notifications, avec plusieurs actions définies (SET_NOTIFICATIONS, ADD_NOTIFICATION, MARK_AS_READ, DELETE_NOTIFICATION, MARK_ALL_AS_READ, etc.). Chaque notification est un objet pouvant contenir un id, un type (voir ci-dessous), un titre, un message, un destinataire, un indicateur de lecture, etc.

Les différents types de notifications sont listés dans `NOTIFICATION_TYPES` – par exemple : **MISSION_ASSIGNED** (mission assignée à un agent, émise par DSI), **ACTION_COMPLETED** (une action terrain terminée par un agent, notifiant DSI+Gouverneur), **PV_GENERATED** (PV généré par le système, notifiant DSI+Gouverneur), **CHANGEMENT_DECLARED** (un changement déclaré par DSI, notifiant le Gouverneur), **MISSION_CREATED** (nouvelle mission créée par DSI, notifiant le Gouverneur), ou **SYSTEM_UPDATE** (messages système génériques pour tous). Ces types permettent de définir des workflows d'alertes selon les rôles.

Le reducer implémente la logique de chaque action : - `SET_NOTIFICATIONS` remplace la liste courante et reset le compteur non lus. - `ADD_NOTIFICATION` ajoute une notification en tête de liste et incrémente le compteur des non lus. - `MARK_AS_READ` marque une notification spécifique comme lue (isRead = true) et décrémente le compteur. - `MARK_ALL_AS_READ` marque toutes comme lues et remet le compteur à 0. - `DELETE_NOTIFICATION` supprime une notification de la liste et recalcule le nombre de non lues restant.

Le state initial contient une liste vide, 0 non-lus, et un indicateur loading.

`NotificationProvider` encapsule ce reducer et fournit les actions via un contexte. À son montage, il charge les notifications initiales en appelant `generateUserNotifications(user)` – une fonction qui crée quelques notifications mock en fonction du rôle utilisateur (par ex. un agent recevra une notification de mission assignée d'emblée). Ceci simule l'état initial.

Le contexte expose des fonctions comme `addNotification`, `markAsRead`, `deleteNotification`, `markAllAsRead` qui dispatchent les actions correspondantes du reducer. Il fournit aussi une méthode `sendNotification(type, data)` pour créer et distribuer une nouvelle notification du type donné aux utilisateurs cibles. Cette fonction détermine les destinataires via `getNotificationRecipients(type, data)` (non détaillée ici), puis pour chaque destinataire, construit un objet notification complet (id unique, titre, message formaté via `getNotificationTitle` / `getNotificationMessage`, infos de sender, etc.) et, si le destinataire est l'utilisateur courant, l'ajoute via `addNotification`. Cela signifie que lorsqu'un événement survient (mission assignée, action terminée…), on peut appeler `sendNotification` et le contexte s'occupera de créer la notification et de l'afficher si elle concerne l'utilisateur actif. De plus, chaque type d'événement a une fonction dédiée pour simplifier son envoi : ex. `notifyMissionAssigned(missionData)` enverra une notification de type MISSION_ASSIGNED à l'agent concerné.

En interne, le contexte utilise la bibliothèque antd pour afficher un toast de notification immédiat (`antdNotification.open`) à chaque ajout de notification, de sorte que l'utilisateur voit instantanément un pop-up en plus de l'ajout dans la liste.

- `src/components/Notifications/NotificationDropdown.js` (et `Notifications/index.js`) : Composant d'interface pour le bouton de notifications dans l'en-tête. Ce dropdown, intégré dans le `MainLayout` (Header), affiche l'icône de notification avec un badge du nombre de non-lus et déroule la liste des notifications à la demande. Bien que le code détaillé du dropdown ne soit pas entièrement cité ici, son fonctionnement repose sur `useNotifications()` pour accéder au contexte. Il affiche sans doute chaque notification avec son titre, son message, et des actions (marquer comme lu, voir plus de détails selon le type). Lorsqu'on clique sur l'icône cloche, on marque possiblement toutes comme lues (ou on ouvre juste la liste).

Grâce au `NotificationContext`, ce composant est mis à jour en temps réel quand des notifications sont ajoutées ou modifiées. L'icône de notification est incluse dans le header du `MainLayout`, juste à côté du badge de rôle et du menu utilisateur.

Layout principal et navigation

- `src/components/Layout/MainLayout.js` : Ce composant définit la structure générale de l'interface une fois connecté, avec la barre latérale (menu) et le header. Il utilise Ant Design Layout (Sider, Header, Content).

Le **menu latéral** (Sider) contient les entrées de navigation principales. Les items sont construits en fonction des permissions de l'utilisateur via `menuItems`. Par exemple, tous les utilisateurs voient *Tableau de bord*, *Carte interactive*, *Missions*. Seuls ceux ayant la permission de créer des actions verront le menu *Actions* (ce qui inclut agents et DSI). Le menu *Changements* est toujours affiché (lecture pour tous). Le menu *Utilisateurs* n'apparaît que pour le DSI (permission canManageUsers), et *Statistiques* seulement pour DSI et Gouverneur (canViewStats). Chaque item a une icône ant-design (DashboardOutlined, EnvironmentOutlined, etc.) pour l'illustration.

Le Sider est **responsive** : si la fenêtre est petite ou sur mobile, le layout passe en mode *drawer* (tiroir) latéral au lieu d'un menu fixe. `MainLayout` gère cela avec un état `isMobile` calculé sur la largeur de l'écran et un écouteur d'évènements resize. Pour mobile, au lieu d'un Sider permanent, on utilise un `<Drawer>` AntD qui peut s'ouvrir/fermer (piloté par `mobileDrawerOpen`). Un bouton de menu

hamburger (MenuFold/Unfold icons) est affiché dans le Header pour ouvrir/fermer le menu mobile. Sur desktop, ce bouton réduit ou étend le Sider (via l'état `collapsed`) sans le masquer entièrement.

En bas du menu latéral, `MainLayout` affiche un petit encart avec l'**utilisateur courant** : avatar avec initiale, nom et commune, et un badge du rôle. Cet encart n'apparaît que si le menu n'est pas collapssé (ou sur mobile dans le drawer). Le badge de couleur utilise `getRoleBadgeColor` pour différencier visuellement les rôles (ex: vert pour Agent, bleu pour DSI, violet pour Gouverneur).

Le **Header** supérieur inclut trois zones : - À gauche, le bouton toggle menu (hamburger) pour mobile ou collapse, puis le titre de la page en cours et un éventuel sous-titre. Le titre de page est déterminé dynamiquement d'après le chemin via `pageTitle` (switch sur `location.pathname`, ex: "/missions/nouveau" => "Nouvelle mission"). Le sous-titre `pageSubtitle` peut indiquer le contexte, par ex. pour un agent sur une page liste on rappelle sa zone (commune/préfecture). Ce sous-titre est notamment utilisé pour les agents d'autorité afin de toujours indiquer *"Zone : [Commune], [Préfecture]"* lorsqu'ils consultent des données locales. Ces textes sont affichés avec des styles adaptés (taille réduite, couleur grisée) sous le titre dans le header. - À droite du Header, on trouve le **NotificationDropdown** (icône de cloche) suivi, sur desktop, d'un badge rappelant le rôle de l'utilisateur. Puis le **menu utilisateur** (avatar et menu déroulant). Ce menu utilisateur est géré via un `<Dropdown>` d'AntD contenant les options *Mon profil*, *Paramètres*, et *Déconnexion*. Cliquer sur "Déconnexion" déclenche directement `logout()` via l'attribut onClick déjà attaché à l'item du menu. Pour profil et paramètres, aucune route dédiée n'est définie (ce sont des entrées prévues mais non implémentées, potentiellement à intégrer plus tard). Sur mobile, le badge de rôle n'est pas affiché (pour gagner de la place, seuls l'avatar et la cloche restent visibles).

Enfin, la section **Content** du layout rend l'Outlet du routeur, c'est-à-dire la page courante. Le Content a un style avec fond gris clair (#f0f2f5, la couleur de fond par défaut AntD) et est scrollable indépendamment (overflow auto). Il est marqué d'un rôle "main" pour l'accessibilité ARIA.

`MainLayout` est donc le squelette UI de toutes les pages authentifiées. Il offre une **expérience utilisateur cohérente** : menu de navigation conditionnel selon permissions, en-tête avec infos utilisateur et notifications, adaptation mobile (drawer), et respect des bonnes pratiques d'accessibilité (clavier, rôles ARIA, libellés).

- `src/components/Layout/Layout.js` : À noter, il existe un fichier `Layout.js` dans le dossier, qui semble définir également un Layout. Il pourrait s'agir d'une ancienne version ou d'une variante. Étant donné que `MainLayout.js` est celui importé et utilisé dans App, c'est lui qui fait foi. On peut supposer que `Layout.js` contenait une logique similaire mais il n'est pas utilisé directement par `App.js`.

- `src/components/Navigation/Navigation.js` : De même, ce fichier suggère une implémentation possible de la navigation (peut-être prévue initialement), mais au final la navigation principale est intégrée dans `MainLayout`. Ce composant n'est pas explicitement monté dans App non plus. Il est possible qu'il ne soit pas utilisé (ou qu'il ait été fusionné dans MainLayout). On n'en tiendra pas compte dans le fonctionnement final puisque le menu est géré via Menu d'AntD dans MainLayout.

Pages de l'application

Les principales pages (sous le dossier `pages/`) sont les suivantes :

- `Login.js` (page de connexion) : Affiche un formulaire de connexion (email/mot de passe) et utilise le contexte d'authentification. À la soumission, il appelle `login(email, password)` depuis `useAuth()`. En cas d'échec, l'erreur est affichée via notification (déjà gérée dans login). En cas de succès, l'utilisateur est redirigé – le routeur, via `ProtectedRoute` et l'état `from`, renverra automatiquement l'utilisateur vers la page qu'il souhaitait voir ou le dashboard. Visuellement, cette page utilise probablement un composant de formulaire AntD et n'a pas de barre de navigation (puisque hors du MainLayout).

- `Dashboard.js` (tableau de bord) : Page d'accueil après connexion. Elle offre une **vue d'ensemble** des chiffres clés et activités récentes. Elle consomme les données via `useMock()` pour obtenir `missions`, `changements` et `users` (utilisateurs) courants, puis filtre ces données en fonction du rôle de l'utilisateur connecté : un agent d'autorité ne voit que les missions/changements de **sa commune**, alors que le DSI ou Gouverneur voit tout (filtrage simple). Ensuite, la page calcule diverses statistiques :

• total de missions, nombre de missions en cours et terminées,
• total de changements, dont combien détectés, en traitement, traités,
• total d'agents (dans la zone filtrée) et surface totale des changements (somme des surfaces signalées).

Elle calcule aussi le *taux de traitement* = pourcentage de changements traités sur l'ensemble (affiché plus loin). Un indicateur de performance (barre de progression) utilise ce taux.

Le rendu de Dashboard comporte plusieurs sections : - Un **message de bienvenue** en haut, via `<Alert>` informatif, personnalisant le texte selon le rôle. Par exemple, pour un agent : *"Bienvenue [nom]. Vous supervisez la zone de [Commune]. Consultez vos missions actives…"*, alors que pour un administrateur : *"Accédez aux données de toute la préfecture Casablanca-Settat…"*. - Une grille de **statistiques principales** en 4 cartes (AntD Statistic) : missions actives (en cours) sur total, changements détectés (non encore traités), surface totale en m² des changements, et taux de traitement en %. Chaque carte a une icône illustrant la métrique (ex: EnvironmentOutlined pour missions, WarningOutlined pour changements détectés, TrophyOutlined pour surface totale, CheckCircleOutlined pour taux) et une couleur distinctive. - Une section **Progression du traitement** affichant une barre de progression globale et un rappel des chiffres *détectés / en traitement / traités* en parallèle. Cela donne un aperçu rapide de l'état d'avancement de la résorption des habitats non réglementaires. - Une section **Actions rapides** avec des boutons permettant d'accéder directement aux sections clés : *Voir la carte interactive*, *Créer une nouvelle mission* (si autorisé), *Enregistrer une action* (si autorisé), *Consulter les statistiques* (si autorisé). Ces boutons sont conditionnés par les permissions (par ex., seul un DSI verra le bouton de création de mission). Cela facilite la navigation vers les fonctionnalités principales depuis le dashboard. - Deux listes d'**activités récentes** : - *Missions récentes* montre les 5 dernières missions créées, avec un bouton "Voir toutes" renvoyant vers la liste des missions. Chaque mission listée est un item avec son titre (cliquable ouvrant la page détails de la mission), un tag de statut coloré (PLANIFIÉE en bleu, EN_COURS orange, TERMINÉE vert), l'icône localisation en avatar, et en description la commune et la date de création relative (par ex. "il y a 5 jours"). Le tag utilise `getStatusColor` pour appliquer la bonne couleur au statut. - *Changements récents* montre les 5 derniers changements détectés, avec un bouton "Voir tous" (visible pour ceux qui peuvent consulter les actions/changements). Chaque item affiche le type de changement (par ex. "EXTENSION HORIZONTALE") en titre, un tag de statut coloré (Détecté rouge, En traitement orange, Traité vert), l'icône Warning en avatar (rouge si détecté non

traité), et en description la commune + surface (ex: “Maârif • 45.5 m²”) et la date de détection relative. - Si aucune mission ou aucun changement récent n’est disponible, un texte secondaire le mentionne. - Enfin, pour les administrateurs (DSI/Gouverneur), une section *Équipe active* affiche quelques stats sur l’équipe : nombre total d’agents, missions terminées, missions en cours, total de missions. Ces chiffres sont présentés dans des Statistic cards avec icônes (TeamOutlined pour agents, CheckCircleOutlined pour missions terminées, ClockCircleOutlined pour missions en cours, etc.).

Si les données sont en cours de chargement (lors du tout premier rendu), un <Spin> avec texte "Chargement du tableau de bord..." est affiché. S’il n’y a aucune donnée du tout (cas extrême : 0 mission et 0 changement), un message d’alerte pourrait s’afficher en bas pour signaler l’absence de données.

En synthèse, la page Dashboard donne un aperçu riche : indicateurs de volume et de progression, dernières actions, et accès direct aux fonctionnalités clés, le tout adapté au contexte de l’utilisateur.

- `Carte.js` (page Carte interactive) : Cette page propose une visualisation géographique des missions et des points signalés (douars/changements). Elle utilise **Leaflet** pour la carte, intégrée manuellement (plutôt que via une dépendance comme react-leaflet) en chargant les assets Leaflet au vol. En effet, un effet `useEffect` dans ce composant charge les fichiers CSS et JS de Leaflet et du plugin de clustering de marqueurs depuis un CDN (via création dynamique de `<link>` et `<script>`). Une fois les scripts chargés, on configure le *fallback* d’icônes (chemins des images de marqueur par défaut) et on marque l’état `mapLoaded` à true.

La page définit des constantes comme le centre initial de la carte (coordonnées lat/long de référence, ici 33.6, -7.4 qui correspond à Casablanca) et le zoom initial. Elle prévoit une limite de points (MAX_MARKERS=100) pour éviter d’en afficher trop simultanément.

Ensuite, lorsque `mapLoaded` passe à true, un second effet initialise la carte : `window.L.map()` est appelée pour créer la carte dans un élément DOM référencé (via `mapRef`). On désactive ou adapte certaines interactions si l’appareil est mobile (ex: `dragging` activé seulement si pas en mobile touch, idem pour doubleClickZoom, etc.) afin d’améliorer l’expérience tactile. On ajoute une couche de tuiles OpenStreetMap comme fond de carte, avec attribution. On place aussi les contrôles de zoom sur la carte (positionnés en bas à droite sur mobile pour ne pas gêner).

Pour l’accessibilité, on attribue au conteneur de carte un rôle application et des instructions ARIA (label "Carte interactive des missions et changements") et on rend la carte focusable au clavier. De plus, un écouteur de clavier est ajouté pour permettre le déplacement de la carte avec les flèches et le zoom avec +/- – un aspect notable en termes d’accessibilité clavier. Sur mobile, un petit message "Utilisez 2 doigts pour déplacer la carte" est même injecté pour informer l’utilisateur du geste requis (puisque le dragging à un doigt est désactivé).

La page utilise les données via `useMock()` : on en extrait `missions` et `changements`. (Ici, **note** : dans `mockApi`, les douars et changements ne sont pas clairement distingués dans `useMock` – on suppose que `changements` contient l’ensemble des points signalés, y compris les *douars signalés par agents* et les *changements détectés par satellite*, bien que dans les données mock ils soient séparés. L’application semble vouloir traiter tous ces **points terrain** de manière unifiée dans la carte).

Un state `filters` est géré pour filtrer les données affichées. Il comprend par exemple un champ `view` pour choisir d’afficher “all” (missions + changements) ou un seul type, un `status` pour filtrer par statut, une `commune`, et éventuellement une plage de dates. Par défaut, pour un agent, on initialise le filtre commune à sa commune afin de ne voir que son secteur.

La fonction de filtrage (`filteredData`) combine les missions et changements en fonction de ces filtres. Par exemple, si `filters.view` est “missions” on ne prend que les missions, si c’est “all” on prend les deux. On filtre par status, commune, et date sur chaque type le cas échéant. Le résultat est un tableau de points géographiques à afficher, où chaque objet se voit ajouter un attribut `dataType` marqué soit `'mission'` soit `'changement'`. Cela permettra de différencier leur affichage (icônes, panneaux d’info).

La page met ensuite en place les **marqueurs** Leaflet : un autre effet (non détaillé plus haut) parcourra `filteredData` et créera soit un cluster de marqueurs (si `clusteringEnabled`) soit des marqueurs individuels. Chaque point est placé à ses coordonnées (extraites via `getSafePosition(item)` qui cherche dans `item.geometry` ou `item.latitude` / `longitude`). On peut deviner que les missions de type POINT ont `geometry.coordinates` défini (lon, lat) et les changements/douars ont directement `latitude` & `longitude`. La couleur ou l’icône du marqueur peut varier selon le type/statut via `getMarkerColor(item)` (non affiché ici, mais probablement missions en bleu, changements en rouge par exemple).

L’interface de cette page inclut probablement : - Un composant **CarteFilters** (filtres) pour permettre à l’utilisateur de définir `filters` (type de points, statut, commune, agent). - Un bouton pour activer/désactiver le clustering des marqueurs (icône de clusters peut-être affichée). - Un bouton pour recentrer ou afficher tous les points. - Un composant **CarteListe** : liste déroulante des éléments affichés (missions/douars), togglable. - Un composant **CartePanel** : panneau latéral qui s’ouvre quand on sélectionne un élément (mission ou douar) sur la carte ou dans la liste, affichant les détails de cet élément.

En effet, le code mentionne l’utilisation de `<Drawer>` d’AntD ou d’un état mobile, et on voit dans les composants qu’il y a CarteListe et CartePanel. On peut donc déduire le comportement : sur la page Carte, l’utilisateur peut cliquer sur un bouton "Liste" pour voir la liste textuelle des éléments filtrés (**CarteListe**), ou cliquer sur un marqueur pour ouvrir un panneau d’information (**CartePanel**).

- `CarteFilters.js` : Ce composant fournit les contrôles de filtres en haut de la carte (ou en overlay). D’après son code, il récupère la liste des agents et des communes via `mockApiService` au montage (getUsers et une méthode utilitaire `getUniqueCommunes`). Il affiche des `<select>` pour choisir :
- *Type :* « Missions + Douars », « Missions seulement », « Douars seulement » (dans le code “douars” semble représenter tous les points terrain hors missions).
- *Statut :* liste déroulante combinant les statuts possibles de missions et de changements (En cours, Planifiée, Terminée, Éradiqué, Détecté, En traitement, Traité). Cela mélange un peu les statuts de natures différentes (par ex. “Éradiqué” concerne plutôt un douar éradiqué, “Détecté/En traitement/Traité” pour changements, etc.). Ce choix permet à l’utilisateur de filtrer globalement peu importe le type de point.
- *Commune :* liste de toutes les communes présentes dans les données (par préfecture).
- *Agent :* liste de tous les agents d’autorité (nom + commune) pour filtrer les points associés à un agent particulier (peut-être les missions assignées à cet agent, ou les changements détectés par un agent).
- Un bouton **Reset** apparaît si un filtre actif est en place, permettant de réinitialiser tous les filtres.

Chaque changement de filtre appelle `onFiltersChange` (prop remontant à Carte.js) avec le nouvel état, ce qui met à jour les points affichés. Ce composant est purement fonctionnel et stylé via un

module CSS (`CarteFilters.module.css`) pour la mise en page (classes comme `filtersContainer`, `filterGroup`, etc.).

- `CarteListe.js` : Cette composant affiche la liste textuelle des éléments actuellement visibles sur la carte, avec un système d'onglets "Missions" et "Douars" (points). Il reçoit en props deux listes filtrées : `missions` et `douars` (douars représentant ici les points terrain, possiblement incluant les changements). Il gère un état interne `activeTab` pour savoir quel onglet est actif.

Pour chaque liste, il génère un bloc de div listant les items. Pour les missions : on affiche le titre de la mission, le statut coloré (couleur déterminée par `getStatutColor`), la commune/préfecture et la date de création, une éventuelle description courte et la liste des agents assignés s'il y en a. Un clic sur un item appelle `onItemSelect(mission, 'mission')` pour signaler à la page parente qu'une mission a été sélectionnée. Pour les douars (points) : on affiche le nom du douar, le statut coloré, la commune/préfecture, les coordonnées géographiques lat/long, éventuellement la mission liée (affiche l'ID de mission), et la date de création du point. Un clic envoie `onItemSelect(douar, 'douar')`. S'il n'y a aucun élément dans une liste, un message "Aucune mission trouvée" ou "Aucun douar trouvé" s'affiche.

Ce composant est présenté en overlay (probablement sur la carte) avec un bouton de fermeture (×) en haut pour le masquer. Il permet à l'utilisateur de parcourir facilement tous les points sans cliquer chaque marqueur sur la carte, ce qui est utile surtout si les points sont nombreux ou trop proches.

- `CartePanel.js` : Le panneau d'informations détaillées pour un élément sélectionné sur la carte. Il reçoit en prop `item` (une mission ou un douar/changement) et affiche ses détails de manière structurée, souvent dans un panneau latéral. Son code gère :
- La récupération des données supplémentaires : via un effet, si l'`item` est une mission, on charge l'agent assigné (getUserById) pour afficher son nom. Par ailleurs, on charge les actions liées à cet item (en appelant `mockApiService.getActionsFiltered({ cibleType: ..., cibleId: ... })`). Ainsi, le panel pourra afficher la liste des interventions (actions) qui concernent ce point ou cette mission.
- Le header du panel affiche un titre comme `"[Mission] Titre de la mission"` ou `"[Changement] Nom du douar ou type de changement"`. Cette distinction est faite via `item.dataType` (valeur injectée dans Carte.js). Il affiche aussi un badge de statut coloré à côté du titre. La couleur et le label du statut sont déterminés par les mêmes fonctions utilitaires que dans la liste (getStatutColor/Label), supportant à la fois les statuts de mission (`status`) et de changement (`statut`).
- La section *Informations générales* : elle liste la zone (commune, préfecture), les coordonnées géographiques si disponibles, puis des champs différents selon s'il s'agit d'une mission ou d'un changement. Pour une mission, on affiche la date de création, la période (date début – date fin) si définie, et les agents assignés. Pour un changement (ou douar), on affiche la surface (m^2) si dispo, la date de détection, et l'agent qui a détecté (par ID, ce serait mieux avec nom mais ici on voit juste "Agent user2" par ex.). Ces champs sont inclus seulement s'ils existent pour l'item.
- Si l'item a une description (missions ont `description`) ou des observations (actions ou changements peuvent avoir `observations`), on affiche un encadré *Description* avec ce texte.
- Si l'item est un changement (point) qui possède des photos *avant/après* (champs `photoBefore` et `photoAfter`), on affiche une section *Photos* avec les deux images côte à côte, légendées "Avant" et "Après". Chaque `<img>` a un onError pour se masquer si la source est invalide (ex. image non trouvée). Cela permet de comparer visuellement l'évolution du site avant/après l'infraction. (*Remarque:* Ces URLs pointent vers le dossier `/images/changements/` du projet, ce qui suggère que quelques images exemple sont disponibles pour illustrer).

- En dessous, la section *Actions liées* liste toutes les actions enregistrées ayant pour cible cette mission ou ce douar. Le contexte Notification a déjà chargé `relatedActions` plus haut. Ici on parcourt `relatedActions` et pour chacun on affiche un bloc : type d'action (avec label lisible via `getActionTypeLabel` – ex: DEMOLITION -> "Démolition") et la date, puis les observations (ou description) de l'action, le montant d'éventuelle amende (ex: "Amende: 25000 DH"), et si l'action a un PV attaché, on indique "PV : Publié" ou "PV : Brouillon" selon son statut. Ces informations donnent un historique des interventions sur ce point.
- Enfin, si l'utilisateur a le droit de modifier (pas en read-only), le panel propose des **actions utilisateur** sous forme de boutons : *Nouvelle action* pour ajouter une intervention sur ce point, *Voir historique* (potentiellement pour ouvrir une page plus détaillée ou un historique complet), et si c'est une mission, *Générer rapport* (peut-être pour créer un rapport/PV global de mission). Ces boutons n'ont pas de handler défini dans ce composant (juste une structure HTML), donc ils semblent prévus pour de futures fonctionnalités ou pour être connectés par la page parente. Le bouton "Nouvelle action" est affiché seulement si `canCreateAction()` est true (or, note: dans AuthContext on n'a pas `canCreateAction` mais `canCreateActions`, cela semble être un petit appel manquant, mais l'intention est claire). Si l'utilisateur est en read-only (par ex. un agent sur une ressource hors de sa commune), aucun de ces boutons n'apparaît.

Le CartePanel se veut donc une fiche synthétique d'une mission ou d'un point terrain, permettant d'enclencher rapidement une nouvelle action si nécessaire. Il complète la vue carte en évitant d'avoir à naviguer vers la page Missions ou Actions séparément pour la plupart des informations contextuelles.

- **Pages Missions** :

- `Missions.js` (liste des missions) – Cette page affiche probablement toutes les missions dans un tableau ou une liste. Elle pourrait ressembler à la page Actions (décrite plus bas) en termes de structure : filtres par statut, commune, recherche, etc., puis listing. D'après App.js, cette page n'est pas protégée par une permission stricte (tous les rôles peuvent la voir via canViewMissions). Un Gouverneur ou DSI verrait toutes les missions, un Agent possiblement seulement les siennes ou celles de sa zone (selon implémentation du filtre). Étant donné la cohérence voulue, il est plausible qu'elle utilise un tableau AntD (<Table>) pour lister missions avec colonnes (Titre, Commune, Statut, Dates, etc.) et actions (voir détail, supprimer si DSI, etc.). Sans le code exact, on peut supposer qu'elle permet également de créer une mission (via bouton *Nouvelle mission* redirigeant vers /missions/nouveau si DSI).

- `MissionDetails.js` – Page de détail d'une mission particulière (accessible via route `/missions/:id`). Elle afficherait plus d'infos sur la mission sélectionnée : son titre, description complète, statut, période, agents assignés (en listant leurs noms via leurs IDs, possiblement grâce aux données `users`), et peut-être la liste des **douars (points) rattachés à cette mission**. En effet, dans les données mock chaque douar a un `missionId` s'il est associé à une mission. Il serait pertinent que MissionDetails affiche la carte des points concernés ou au moins leur liste, ainsi que les actions effectuées dans le cadre de cette mission. Il pourrait même y avoir un bouton "Voir sur la carte" pour ouvrir la vue map centrée sur cette mission. Sans code, difficile de détailler, mais on peut imaginer que cette page utilise certains composants du CartePanel (ou du moins une logique similaire) pour lister les actions liées à la mission et peut-être un lien pour générer un rapport global.

- `NouvelleMission.js` – Formulaire de création d'une mission (accessible uniquement au DSI). Ce formulaire demanderait un titre de mission, une description, la zone géographique (préfecture/commune), éventuellement un type (point ou zone), une période (date début/fin), et quels agents sont assignés. Il est possible que ce formulaire intègre un composant de sélection

de zone sur une carte (d'où l'existence de `SelecteurZone`). Par exemple, pour une mission de type zone (inspection d'un quartier complet), l'utilisateur DSI pourrait dessiner un polygone ou sélectionner une commune. Le code `SelecteurZone.js` probablement permet de choisir une zone plus précise (peut-être via Leaflet dessin). `NouvelleMission` pourrait donc d'abord faire appel à `SelecteurZone` (soit incorporé, soit via modale) pour définir la géométrie de la mission. Une fois le formulaire soumis, la mission serait créée via `mockApiService.createMission`, mise à jour du contexte etc., puis redirection vers la liste des missions ou la page détail.

- `SelecteurZone.js` – Peut-être un composant/page qui affiche une mini-carte Leaflet permettant de sélectionner une zone (par exemple en cliquant pour mettre un marqueur ou en dessinant un polygone). Il serait utilisé lors de la création d'une mission ou d'un changement pour géolocaliser précisément la zone d'intervention. N'ayant pas son code détaillé, on suppose qu'il utilise Leaflet de manière similaire à NouvelleAction (qui fait placement de point).

- **Pages Actions** : L'ensemble de fichiers dans `pages/Actions/` gère les interventions terrain (actions effectuées par les agents).

- `Actions.js` – Page listant toutes les actions enregistrées. Elle utilise `useMock()` pour récupérer `actions` et `users`. Comme pour d'autres pages, on combine éventuellement les données mock avec `mockData` en fallback si l'API retourne vide (ici `baseActions` prend `apiActions` ou sinon les actions du fichier mockData). Un filtrage par rôle est appliqué : si l'utilisateur est un Agent, on filtre la liste pour ne montrer que les actions **qu'il a réalisées ou qui concernent sa commune**. Ainsi un agent d'autorité ne voit pas toutes les actions de la préfecture, seulement les siennes et celles de son secteur, conformément à son périmètre.

  Ensuite, plusieurs états de filtre dans le composant : `searchText` pour une recherche textuelle, `selectedType` (filtrer par type d'action, ex: démolition, signalement…), `selectedStatus` (filtrer par statut d'action si utilisé, par ex. Terminée vs En cours), et potentiellement `dateRange` (intervalle de dates). Un effet `useEffect` applique ces filtres sur la liste d'actions filtrée par rôle : recherche textuelle sur le champ observations, commune, ou nom d'agent ayant effectué l'action (on retrouve l'agent via son userId pour matcher le nom), puis filtrage par type si choisi, par statut si choisi, et par date si un intervalle est sélectionné. Le résultat est stocké dans `filteredActions` et sert de source de données à afficher.

  La page calcule aussi quelques statistiques rapides sur ces actions filtrées : total affiché, combien de démolitions, combien de signalements, et combien avec PV attaché.

  Côté rendu, on a : - Un en-tête avec le titre "Actions sur le terrain" et un sous-texte indiquant s'il s'agit des actions de la commune (pour un agent) ou de toute la préfecture (pour DSI/Gouverneur). Si l'utilisateur est Agent, on ajoute par ex. "*Actions de [Ma Commune]*" sinon "*Toutes les actions de la préfecture*". - Un bouton "Nouvelle action" apparaît à droite de ce header uniquement pour les agents (puisqu'eux et les DSI peuvent créer des actions, mais ici il a été conditionné à role Agent). En cliquant, il navigue vers `/actions/nouveau` (le formulaire de création d'action). - Un tableau de **statistiques** sous forme de 4 cartes (AntD Statistic) montrant : total actions, nombre de démolitions, nombre de signalements, nombre ayant un PV. Ces chiffres se basent sur `stats` calculé plus haut. Cela donne une vue rapide de la répartition des actions affichées. - Un bloc de **filtres** avec : - une barre de recherche textuelle (placeholder "Rechercher (obs., commune, agent)…") liée à `searchText`, - un select de type d'action alimenté par `ACTION_TYPES` (qui contient les labels "Démolition", "Signalement", "Non-

démolition”), - un select de statut (ici les statuts d’actions définis semblent être "EN_ATTENTE", "EN_COURS", "TERMINEE" – possiblement utilisé si on distingue actions en cours ou terminées). - (Le code mentionne aussi un RangePicker date dans l’import, mais il est commenté/désactivé dans l’UI actuelle d’après la variable dateRange non utilisée dans JSX, sauf s’ils l’ont omis). - Le **tableau des actions** lui-même, rendu via `<Table>` Ant Design. Les colonnes définies sont : - *Date* (affiche la date formattée jour/mois/année, triable), - *Type* (affiche un `<Tag>` coloré du type d’action, couleurs définies dans ACTION_TYPES, et permet un filtrage via un menu intégré au tableau), - *Agent* (affiche le nom de l’agent ayant fait l’action, calculé par une fonction `agentName(a)` qui utilise l’ID pour retrouver le nom dans la liste des utilisateurs), - *Commune*, - *Observations* (affiche le texte des observations avec ellipsis si trop long, tooltip au survol), - *PV* : dans cette colonne, si l’action a un PV associé (probablement `action.pv` non null), on affiche un bouton "Voir PV" cliquable qui ouvre la modale de PV (voir plus bas). Ce bouton est actif seulement si `canViewPV(user, action)` est vrai – c’est une fonction utilitaire qui reprend la logique de AuthContext : un DSI/Gouverneur peut voir tous les PV, un agent peut voir les PV de ses propres actions ou de sa commune. Si pas de PV ou pas autorisé, on affiche un tiret. - *Actions* (colonnes d’actions utilisateur sur chaque ligne) : pour un utilisateur en lecture seule (ex: un agent voyant l’action d’un autre agent hors de sa zone, ou un Gouverneur sans pouvoir éditer) on met juste un tiret. Sinon, on pourrait afficher des boutons "Modifier" et "Supprimer". Dans le code, ces boutons n’ont pas de onClick, ils sont indicatifs (peut-être pour futur, ou bien l’édition/suppression n’est pas implémentée dans le mock). Le DSI ou l’agent propriétaire d’une action pourrait cliquer "Modifier" pour ouvrir un formulaire pré-rempli (non implémenté réellement), ou "Supprimer" pour la retirer.

Si les données sont en cours de chargement initial et qu’aucune action n’est encore disponible, on affiche un Spin de chargement central. Si le tableau est vide après filtrage, on affiche un message d’alerte "Aucune action ne correspond aux critères".

Enfin, la page gère une modale de visualisation de PV (`pvModalVisible`). Lorsque l’utilisateur clique "Voir PV" sur une action, `handleViewPV(action)` met à jour `selectedAction` et ouvre la modale si l’action a un PV et que l’utilisateur peut le voir. La **modale PV** a pour contenu soit le texte du PV soit un message d’indisponibilité. Ici, on affiche le contenu du PV en préservant les sauts de ligne (`<div style={{ whiteSpace: 'pre-line', fontFamily: monospace }}>`) : on prend soit `selectedAction.pv.template` soit `selectedAction.pv.contenu` ou une phrase par défaut. Les données mock créent un champ `pv` dans l’action possiblement lorsque `createPV` est appelé, contenant soit un template texte soit un lien. Dans notre cas de mock, on insère directement le texte intégral du PV dans `mockData.pvs` et on pourrait l’associer via l’id d’action. Quoi qu’il en soit, le modal permet de **consulter le procès-verbal** lié à une action, ce qui est crucial pour les Gouverneurs par exemple (ils valident les PV). La modale propose un simple bouton "Fermer". S’il y avait besoin d’éditer, ce serait via la page PVEditor séparée.

- `NouvelleAction.js` – Il s’agit de la page de création d’une nouvelle action (accessible par /actions/nouveau). C’est un composant relativement complexe qui offre un **formulaire complet et une carte de sélection de position**. Contrairement à d’autres formulaires plus simples, ici l’interface mélange du formulaire AntD classique et une carte Leaflet pour choisir un point GPS de l’action.

D’entrée, on remarque qu’il importe beaucoup de choses : des icônes (AimOutlined, etc.), des hooks de route (`useNavigate`), `useAuth`, `mockApiService`, des utilitaires (ACTION_TYPES, createAutoPV, generateActionId, geolocationHelpers), et des composants d’upload `ActionPhotoUploader`, etc. Cela

indique que la page va gérer : - Le choix du type d'action (démolition, signalement, non-démolition), - La saisie de la date et observations, - Le montant de l'amende si applicable, - La décision de créer un PV immédiatement ou non, - L'upload des photos avant/après pour une démolition, - La sélection de la localisation sur une carte (lat/long).

Le composant utilise le formulaire AntD via `Form.useForm()`. Quelques states locaux : - `requiresAmende` (booléen indiquant si le type d'action sélectionné nécessite de saisir une amende – dans ACTION_TYPES, il semble que *Non-démolition* et *Signalement* ont requiresAmende: false, *Démolition* est marqué false par défaut aussi dans le code, mais on le force plus loin), - `loading` (chargement lors de la soumission), - `geoloading` (pour la géolocalisation via GPS), - `createPvNow` (booléen, case à cocher pour créer tout de suite le PV brouillon – initialisé à true), - `uploadedPhotos` et `photosValid` pour la gestion des photos avant/après.

Côté carte : - On charge Leaflet de manière dynamique comme dans Carte.js (insère le CSS/JS si pas déjà présent). - Une fois Leaflet chargé (`leafletLoaded`), on initialise une petite carte (centre sur MAP_CENTER = [33.6, -7.4], zoom 11) sur le ref `mapRef`. On ajoute un contrôle de zoom, la couche OSM, etc. On attache un handler `onMapClick` qui, si on est en mode sélection (`isSelectingLocation` true), place un marqueur à l'endroit cliqué via `setPointOnMap(lat, lng)`. On stocke la référence de la carte dans `mapInstanceRef` pour réutilisation. On nettoie aussi le marker et l'event listener à la fermeture du composant pour éviter les fuites. - La fonction `setPointOnMap(lat, lng, source)` place un marqueur Leaflet custom (un petit point bleu numéroté "1") sur la carte à la position donnée. Elle centre la vue dessus, met à jour l'état `selectedPosition` (avec lat, lng, source), met fin au mode sélection (`isSelectingLocation=false`), et remplit les champs hidden latitude/longitude du formulaire avec ces valeurs. Un message de succès notifie que la position est enregistrée en affichant les coordonnées formatées. Si un marqueur existait déjà, on le remplace (la fonction `clearMarker` supprime le précédent). - L'utilisateur peut cliquer un bouton "Sélectionner sur carte" pour entrer en mode sélection (active `isSelectingLocation` et affiche une notification info "Cliquez sur la carte pour choisir l'emplacement"). Un bouton "Utiliser GPS" appelle `handleGetCurrentLocation()` qui utilise `getCurrentPositionWithUI` (un helper encapsulant `navigator.geolocation`) – cela affiche probablement un message de chargement "Localisation GPS en cours…" puis, si succès, appelle `setPointOnMap` avec les coordonnées GPS obtenues. En cas d'échec, on log l'erreur. Ce bouton est très utile sur le terrain pour permettre à un agent de prendre sa position actuelle. - Un bouton "Changer" (avec icône poubelle) permet de réinitialiser la position sélectionnée (supprime le marker, vide lat/long du formulaire).

Sur la partie formulaire : - Au chargement, on préremplit certains champs si possible : la date (par défaut date/heure actuelle), et la commune/préfecture de l'utilisateur pour ne pas avoir à les resaisir (utile si l'agent ne gère qu'une commune). On coche par défaut "Créer PV" à true. - Le formulaire comprend plusieurs sections (découpées en Card pour l'esthétique) : - **Informations de base** : Type d'action (Select alimenté par ACTION_TYPES), Date d'intervention (DatePicker date+heure), Observations (TextArea), Montant de l'amende (champ numérique) qui n'apparaît que si `requiresAmende` est true (par ex. pour Démolition), et la case à cocher "Créer le PV immédiatement" (champ `createPV`) qui est cochée par défaut. Cette case a un tooltip expliquant qu'un PV brouillon sera créé et qu'on sera redirigé vers son édition. - **Documentation (photos)** : Un composant `<ActionPhotoUploader>` est rendu. Ce composant spécialisé permet d'uploader deux photos (avant et après). Il repose sur `DocumentUploader` mais configuré pour photos avant/après (voir section Composants communs). L'attribut `required={requiresAmende}` signifie peut-être qu'on exige les photos si une amende est nécessaire (typiquement en cas de démolition, on veut photos avant/après). Le composant gère l'upload (voir plus bas) et renvoie les fichiers via `onChange`. Le code met ensuite un indicateur de combien de

photos ont été ajoutées et si elles forment bien une paire avant/après complète. Par exemple, s'il manque une des deux photos en cas de démolition, on affiche un warning "⚠ *1 photo (compléter avant/après)*". Si les photos sont complètes, on affiche un check vert avec le nombre de photos. - **Localisation** : Champ Préfecture et Commune (sélecteurs avec les valeurs – ici on voit seulement Casablanca-Settat et Mohammedia en préfecture, et quelques communes en exemple). Puis les champs Latitude et Longitude (en lecture seule, remplis par la sélection sur la carte). Ces champs sont requis (on doit avoir une position). Enfin une carte est affichée sous ces champs dans la section "Emplacement d'intervention" : on montre soit une Alert verte "Emplacement sélectionné: X,Y" si c'est fait, soit une Alert info/avertissement demandant de sélectionner ou indiquant que la sélection est en cours. Ensuite, on affiche les boutons de contrôle de localisation (rendus par la fonction `renderLocationControls()`): tant qu'aucune position n'est choisie, on montre les boutons "Sélectionner sur carte" (ou "Cliquez sur la carte" si déjà en cours) et "Utiliser GPS". Une fois un point choisi, on montre un tag vert "Emplacement sélectionné" et un bouton "Changer" pour réinitialiser. Puis la div contenant la carte Leaflet (`<div ref={mapRef} style={…}></div>`) avec style hauteur 400px, etc.. S'il y a un délai pour initialiser la carte, un message "Initialisation de la carte..." s'affiche au centre. Enfin, en dessous de la carte, un petit texte d'aide mentionne "*Astuce : Cliquez "Sélectionner sur carte" puis sur la carte, ou utilisez "Utiliser GPS". Un seul point est enregistré.*" pour guider l'utilisateur.

- **Soumission du formulaire** (`handleSubmit`) : Quand l'utilisateur valide, la fonction récupère `values` du formulaire AntD. Elle effectue plusieurs validations manuelles en plus de celles gérées par AntD et Zod :
  - Si aucune position n'a été sélectionnée (`!canSubmit`, c.-à-d. pas de selectedPosition), on affiche une erreur "Emplacement manquant".
  - On valide les coordonnées via `validateCoordinates(lat, lng)` (vérifie -90<=lat<=90 etc.). Si invalide, on notifie une erreur "Coordonnées invalides".
  - Si le type est *DEMOLITION*, on exige deux photos avant/après : on utilise `validateBeforeAfterPhotos(uploadedPhotos)` pour vérifier qu'on a bien une photo catégorie 'before' et 'after'. Si non, on notifie l'utilisateur qu'il doit fournir les photos nécessaires.
  - Une fois ces pré-conditions remplies, on procède à la composition de l'`actionPayload` à envoyer :
  - On génère un identifiant d'action unique via `generateActionId()`.
  - On crée un champ `geometry` de type Point à partir de `selectedPosition` (lon/lat).
  - On gère l'upload effectif des photos s'il y en a : on extrait les fichiers des objets `uploadedPhotos` et on appelle `defaultUploadManager.handleFilesUpload(files, { category: 'action_photos', metadata: {...}, showNotifications: false })`. Ce defaultUploadManager (dans utils/uploadHelpers) simule sans doute un envoi vers un serveur ou stocke les fichiers localement. Le résultat `uploadResult.results` fournit un tableau d'objets avec id, url, nom original, etc., qu'on stocke dans `uploadedFiles`. On entoure d'un try/catch pour ne pas interrompre en cas d'erreur d'upload (juste on log un warning).
  - On complète ensuite l'objet `actionPayload` avec :
    - `type` (valeur du formulaire),
    - `dateAction` (values.date converti en objet Date),
    - `commune` et `prefecture` (du formulaire, préremplies normalement),
    - `userId` l'ID de l'utilisateur courant, et on embarque aussi un sous-objet `user` avec id, name, email de l'agent pour éviter de le rechercher plus tard,
    - `observations` texte,

        - `montantAmende` (ou null si non applicable),
        - la location { lat, lng, source } et un champ `geoMetadata` avec timestamp (on garde ces infos de géoloc),
        - la liste des `photos` uploadées (on utilise `uploadedFiles` calculé pour stocker id, url, etc. de chaque photo).
    - On appelle `await mockApiService.createAction(actionPayload)` pour simuler la sauvegarde de l'action. Cela va ajouter l'action en mémoire dans la liste `actions` du service et déclencher notifyListeners (donc via `useMock` toute liste d'actions se mettra à jour).
    - Si l'utilisateur a choisi de créer un PV immédiatement (`values.createPV` true, ou le state createPvNow true par défaut), on essaie de générer le PV :
        - On construit `pvData` en utilisant `createAutoPV(createdAction, null, user, null)` pour obtenir un objet de contenu de PV auto. Ici, on passe `mission` et `cible` null parce qu'une action peut être liée soit à une mission soit à un douar ; dans notre usage, on a ciblé potentiellement un douar, mais l'exemple choisit null (on pourrait enrichir plus tard). `createAutoPV` va produire un brouillon de PV avec date, nom agent, commune, préfecture, etc. rempli depuis l'action. On fixe le statut du PV à 'BROUILLON'.
        - On appelle `mockApiService.createPV(pvData)` pour stocker ce PV dans la liste des PV mock. Le service `createPV` associe le PV à l'action correspondante (il cherche l'action par id et l'inclut dans l'objet PV) et génère un numéro de PV unique (via `generatePVNumber`).
        - Si la création du PV réussit, on notifie succès "*Action + PV créés*" et on inclut un lien cliquable vers le PV fraîchement créé (route `/actions/pv/[id]`). On redirige même automatiquement (`navigate`) l'utilisateur vers la page d'édition du PV (`/actions/pv/:pvId`) pour qu'il puisse éventuellement compléter ou valider le PV immédiatement.
        - Si la création du PV échoue (peu probable en mock, sauf erreur), on notifie que l'action est créée mais le PV non, avec un message d'avertissement.
    - Si l'utilisateur avait décoché *Créer PV*, alors après création de l'action on notifie simplement "*Action créée*" et qu'il pourra générer le PV plus tard depuis la liste.
    - Enfin, si on n'a pas déjà navigué vers le PV, on navigue de retour vers `/actions` (la liste) pour finir le cycle.

  Pendant tout ce processus `setLoading(true)` en début, et dans le finally on fait `setLoading(false)` pour réactiver le bouton.

- Si l'utilisateur annule via le bouton Annuler, on utilise `navigate('/actions')` pour retourner à la liste.

Le composant `NouvelleAction` est donc très complet : il combine un **formulaire** validé (AntD form + validation custom) et une **carte interactive** pour géolocaliser l'action. Il fait appel à de nombreuses fonctions utilitaires pour réduire le code (génération d'ID, de PV auto, upload manager, etc.). C'est un bon exemple de l'architecture du projet où on utilise les contextes (Auth pour obtenir `user`, MockApi pour créer l'action/PV), les composants partagés (Upload, etc.), et Ant Design pour la cohérence visuelle.

- `ActionCreate.js` – Ce fichier existe dans `pages/Actions`, mais il est possible qu'il ne soit pas vraiment utilisé (on voit qu' `App.js` déclare une route /actions/create vers ActionCreate, enveloppée de ProtectedRoute). Peut-être est-ce un duplicat de NouvelleAction ou une ancienne version (le terme anglais "Create" vs français "NouvelleAction" laisse penser à un composant

initial non francisé). Vu la présence de NouvelleAction qui semble plus complète, ActionCreate.js est possiblement obsolète ou sert à la création d'une action d'un autre type (peut-être la déclaration d'une *action particulière* comme une contravention ou un PV hors champ?). Sans plus d'info, on peut ignorer ce doublon ou signaler qu'il existe mais que `NouvelleAction` est la page principalement utilisée pour créer une action terrain.

- `PVEditor.js` – Importé comme `PVEditor` dans App (route `/actions/pv/:pvId`). Ce composant serait la page d'édition détaillée d'un PV. Lorsqu'un PV est créé automatiquement en brouillon, on redirige l'agent ici pour qu'il le complète. `PVEditor` n'est pas dans le dossier `pages/PVViewer` (il y a un `PVViewer.js` mentionné dans la structure), mais plutôt dans `pages/Actions/PVEditor.js`. Il pourrait contenir un formulaire avec le contenu du PV fractionné en champs éditables (constat, décision, sanction, etc.), ou juste afficher le PV complet avec possibilité de le valider. Étant enveloppé de ProtectedRoute canCreateActions, un agent peut y accéder pour ses PV, tout comme DSI ou Gouverneur (eux aussi ont canCreateActions d'après AuthContext). Une amélioration serait de n'autoriser que DSI/Gouverneur à valider (peut-être via un bouton "Valider le PV" visible seulement pour Gouverneur). Sans son code, on imagine que PVEditor récupère via `useMock()` le PV correspondant à l'ID dans l'URL (ex: en combinant `actions` et `pvs` ou via un appel direct type getPVById). Ensuite, il affiche les détails du PV. Comme `mockData.js` contient des exemples de contenu textuel de PV multi-lignes, l'éditeur pourrait présenter ces champs remplis. Il pourrait y avoir un bouton "Publier le PV" qui change son statut de BROUILLON à VALIDE/PUBLIÉ via `mockApiService.validatePV(id, validatorUserId)`. Ce dernier existe dans le service mock (il marque statut VALIDE, enregistre validatedAt et validatedBy). En somme, PVEditor est prévu pour finaliser un PV : c'est l'étape où le supérieur hiérarchique revoit et valide le rapport.

- **Pages Changements** :

- `Changements.js` – Similaire aux pages Missions/Actions, mais pour lister les changements détectés (ex: par satellite). Elle serait visible par tous (role Agent voit ceux de sa zone, etc.). On peut s'attendre à un tableau listant chaque changement avec type, commune, statut (Détecté, En traitement, Traité), surface, date, peut-être photo avant/après (peut-être via un bouton "Comparer photos"). Le DSI aurait un bouton "Déclarer un changement" pour en ajouter manuellement (par exemple si un changement a été repéré ailleurs qu'à travers le système automatique, ou pour simuler l'arrivée d'une donnée externe).
- `ChangementCreate.js` – Formulaire de déclaration d'un nouveau changement (accessible par `/changements/nouveau`, DSI uniquement). Ce formulaire serait proche de NouvelleAction mais sans la partie action. Il s'agirait de saisir :
    - Type de changement (enum : Extension horizontale, Extension verticale, Nouvelle construction, etc.),
    - Description, surface, date de détection, éventuellement télécharger les deux photos (avant/après) s'ils sont fournis par le satellite,
    - Localisation du changement sur la carte (similaire à sélection d'un point, ou peut-être associer à un douar existant),
    - Ce formulaire créerait en base un nouvel entry dans `changements` (via mockApiService.createChangement). Dans `mockApi`, createChangement est implémenté de façon analogue aux autres, en créant un id et en stockant le changement.
    - Après création, peut-être notifier le Gouverneur (via NotificationContext.notifyChangementDeclared par exemple).

N'ayant pas le code source sous les yeux, ceci est spéculatif, mais cohérent avec le reste du système.

- `Utilisateurs.js` (page Gestion des utilisateurs) : Réservée au Membre DSI (admin). Elle liste tous les utilisateurs actuels (du moins ceux dans `mockData.users`) avec leurs informations : nom, email, rôle, commune, téléphone, date de création, etc. Elle pourrait permettre :

- d'ajouter un nouvel utilisateur (via un bouton ou le composant `UserCreateButton.js` fourni),
- de modifier le rôle ou les infos d'un utilisateur (peut-être via une ligne de tableau éditable ou un bouton "Modifier"),
- de supprimer un utilisateur.

`UserCreateButton.js` peut être un composant qui ouvre un modal de création de compte (il est mentionné dans le dossier). La page utilisateurs intégrerait ce bouton dans son header.

Vu l'enjeu de sécurité, la suppression ou modification pourrait demander confirmation.

Cette page utiliserait `useMock().users` pour les données, et les méthodes du service (createUser, updateUser, deleteUser) pour les opérations (on voit que MockApiService implémente ces méthodes CRUD utilisateurs). Par exemple, `createUser` crée un id unique et ajoute l'utilisateur dans la liste. Les rôles disponibles seraient AGENT_AUTORITE, MEMBRE_DSI, GOUVERNEUR – on doit empêcher d'avoir plusieurs DSI ou Gouverneurs s'il y a des contraintes, mais en mock ce n'est pas géré.

- `NotFound.js` : Page très simple pour les routes non reconnues (404). Elle affiche un message du genre "Page non trouvée" avec peut-être un lien pour revenir à l'accueil. Étant donnée son importation dans App pour la route * (wildcard), elle sera rendue si l'utilisateur navigue vers une URL inconnue. Souvent, on utilise le composant `<Result status="404" ...>` d'AntD pour un bel affichage.

Composants utilitaires communs

En plus des pages et gros composants décrits, le projet contient quelques **composants communs** notables :

- `FileUpload/DocumentUploader.js` : Composant générique d'upload de fichiers avec interface drag & drop (basé sur `Upload.Dragger` d'Ant Design). Il accepte des props pour configurer le type de fichiers permis (`fileType` : PHOTOS, PDF ou ALL) avec une liste d'extensions acceptées et une taille max définies en haut. Selon le `context` d'utilisation (défini via `UPLOAD_CONTEXTS`, par ex. ACTION_PHOTOS, CHANGEMENT_DOCS, PV_ANNEXES) il pourrait ajuster certains comportements, bien que dans le code fourni ces contextes ne semblent pas beaucoup moduler la logique hormis labellisation.

Ce composant gère la validation des fichiers avant upload : `beforeUpload(file)` vérifie extension et taille, refuse si non conforme. Il retourne `false` pour empêcher l'upload auto par AntD, car ici on souhaite contrôler nous-mêmes l'envoi (on attend que l'utilisateur soumette le formulaire global, puis on utilise le `defaultUploadManager`).

Lorsqu'on sélectionne un fichier (ou glisse/dépose), AntD passe un objet `fileList`. La fonction `handleChange({ fileList: newFileList })` est appelée : elle parcourt `newFileList` et pour chaque fichier image, génère immédiatement un aperçu (en DataURL via FileReader) pour pouvoir l'afficher dans la liste. Puis elle met à jour l'état local `fileList` et appelle `onChange` prop pour

remonter la nouvelle liste de fichiers au parent. Les fonctions `handlePreview(file)` ouvrent un aperçu : si image, on ouvre une modale interne pour la voir en grand (en stockant `previewImage` et en rendant un `<Modal>` avec l'image). Si PDF, on ouvre un nouvel onglet avec l'URL du PDF. La fonction `handleRemove(file)` supprime un fichier de la liste et met à jour le parent aussi.

La fonction de rendu renvoie soit un double upload "avant/après" si `showBeforeAfter` est true, soit un upload standard unique/multiple. - En **mode standard** (utilisé pour pièces jointes ou photos multiples hors distinction) : on utilise un seul `<Dragger>` AntD avec fileList complet, accept, etc., qui affiche une zone "Cliquez ou glissez vos fichiers" et liste tous les fichiers sélectionnés en dessous. - En **mode Avant/Après** (utilisé pour ActionPhotoUploader) : le composant rend deux colonnes (`<Col span={12}>` chacune) avec chacune un `<Card>` intitulé "Photo AVANT" et "Photo APRÈS". Dans chaque carte, un `<Dragger>` est affiché, filtrant `fileList` pour ne montrer que les fichiers dont `file.category === 'before'` dans la première colonne et ceux avec `category === 'after'` dans la seconde. Cela permet de séparer visuellement les deux photos. Cependant, il faut noter que dans la version actuelle du code, la catégorie n'est pas automatiquement attribuée aux fichiers sélectionnés. Peut-être s'attend-on à ce que l'utilisateur ajoute explicitement une photo dans la zone "Avant" (ce qui pourrait taguer le fichier en category 'before') puis une photo dans la zone "Après" (tag 'after'). Le code n'indique pas comment `category` est défini, c'est possiblement implémenté dans une version ultérieure ou manuellement via le composant d'upload (on voit dans le JSX qu'ils ne distinguent pas onChange entre les deux Dragger – ils partagent handleChange, ce qui laisse penser que l'utilisateur doit ajouter un fichier dans chaque zone séparément pour qu'ils soient séparés).

En tout cas, `DocumentUploader` fournit la base pour le composant suivant.

- `FileUpload/PhotoComparer.js` : Ce composant sert à afficher côte à côte (ou superposées) deux images avant/après avec des contrôles. Il est probablement utilisé soit dans CartePanel (pour les photos, bien qu'ils aient fait plus simple en affichant juste deux `<img>`), soit prévu pour PVEditor ou Stats. `PhotoComparer` importe divers outils d'affichage (Modal, Tag, Tooltip, etc.) et offre possiblement un mode plein écran, un bouton pour swapper les images, et un slider pour comparer (classique effet "avant/après"). Sans entrer dans les détails, c'est un widget d'UX avancé. Il prend en props `photoBefore`, `photoAfter` et affiche un titre "Comparaison Avant/Après". Il offre éventuellement un bouton plein écran (`allowFullscreen`) pour voir en grand, et des métadonnées (peut-être la date ou l'auteur de chaque photo).

- `FileUpload/FilePreview.js` : Non détaillé, probablement un petit composant pour afficher la vignette et le nom d'un fichier, possiblement utilisé dans DocumentUploader pour la liste des fichiers.

- `components/ActionForm/ActionForm.js` : Il convient de mentionner ce composant isolé car il contient du code proche de `NouvelleAction` mais plus générique. En effet, `ActionForm` utilise React Hook Form et Zod pour valider un formulaire d'action. C'était peut-être une tentative de migrer le formulaire d'action vers react-hook-form (une autre approche que AntD Form). On y trouve un schéma Zod `actionFormSchema` définissant des validations fines (type d'action requis, cibleType requis, cibleId non vide, date <= aujourd'hui, observations longueur min 10, montantAmende nombre entre 100 et 100000 si présent, latitude/longitude entre -90/90 et -180/180). Il a même des `.refine` pour exiger montantAmende si type=DEMOLITION, et pour exiger que si une coordonnée est fournie, l'autre aussi. Ce composant semble être un formulaire d'action universel (peut-être pour être utilisé dans une modale ou pour éditer une action existante). Il gère du localStorage pour sauver un brouillon toutes les 2s (`saveDraft`, `loadDraft`, `clearDraft`, etc.) – fonctionnalité intéressante pour

éviter de perdre des données saisies. Il propose la géolocalisation (`handleGetCurrentLocation` qui reprend le même getCurrentPositionWithUI que plus haut), l'upload de fichiers (`handleFilesChange` qui utilise validateBeforeAfterPhotos pour vérifier la paire), la sélection de cible mission ou douar (`handleCibleChange` préremplit lat/long si une mission est choisie), la génération d'un PVPreview en direct (`generatePVPreview` qui utilise `createAutoPV` pour produire un objet PV et le stocke dans pvPreview), etc. Il y a un aperçu du PV brouillon mis à jour en temps réel via un effet (dès que form valid et type+cible présents).

En somme, `ActionForm` est un composant très complet et sophistiqué qui encapsule la logique de création d'action. Il est sans doute utilisé (ou prévu de l'être) dans une modale contextuelle (peut-être `MissionCreationModal` ou autre). Toutefois, dans la structure actuelle, la page `NouvelleAction` a sa propre implémentation. Il est possible que l'équipe ait deux approches en test : l'une avec AntD Form (NouvelleAction) et l'autre avec React Hook Form + Zod (ActionForm), et décidera laquelle garder. **Pour l'instant, la page NouvelleAction est fonctionnelle.**

Quoi qu'il en soit, le code de `ActionForm` montre l'intention d'une expérience améliorée : sauvegarde de brouillon local en continu, prévisualisation automatique du PV au fur et à mesure, etc., ce qui sont d'excellentes idées pour l'utilisateur.

- `MissionCreationModal.js` : Probablement un composant contenant un `<Modal>` AntD avec un formulaire pour créer une mission rapide. Il est cité dans `components`, on peut supposer qu'il utilise un composant Form interne ou le `SelecteurZone` pour choisir un périmètre. Peut-être est-il appelé depuis un bouton du Dashboard (ex: "Créer mission" ouvre la modale plutôt que de naviguer sur une page séparée). Le code n'a pas été examiné ici par manque de temps, mais sa présence suggère une alternative à la page NouvelleMission.

- `UserCreateButton.js` : Comme évoqué, c'est sans doute un bouton (peut-être dans la page Utilisateurs) qui ouvre une modale de création d'un nouvel utilisateur (un petit formulaire demandant nom, email, rôle, commune, etc.). Il peut aussi directement intégrer un formulaire inline ou simplement être un `Button` qui appelle `mockApiService.createUser` après avoir collecté les infos (peu probable sans UI intermédiaire). À clarifier selon utilisation.

Services et données (complément)

Le service `mockApi.js` mérite un petit résumé du contenu de `mockData.js` car cela éclaire les fonctionnalités attendues : - **Préfectures & Communes** : la liste des préfectures de Casablanca (Casablanca-Anfa, Ain Chock, etc., plus Mohammedia) et un dictionnaire des communes par préfecture sont définis dans `mockData`, plus des fonctions helpers comme `isCommuneInPrefecture` pour vérifier cohérence. Ceci laisse penser que le formulaire de création de mission ou d'utilisateur utilise ces listes déroulantes. - **Utilisateurs (users)** : une liste d'utilisateurs fictifs est fournie avec différents rôles (membre DSI, gouverneur, et plusieurs agents). Chaque utilisateur a id, nom, email, rôle, commune, préfecture, téléphone, date création. Cela permet de tester l'appli avec différents logins. Par exemple *membredsi@auc.ma* est MEMBRE_DSI, *gouverneur@…* est GOUVERNEUR, *user1@casablanca.ma* est un agent de Maârif, etc. Les mots de passe ne sont pas stockés là (le mock login accepte n'importe quel password non vide avec email existant). - **Douars (points signalés par agents)** : Liste de quelques douars avec id, nom, statut (SIGNALE, EN_COURS_TRAITEMENT, ERADIQUE), localisation (latitude/longitude), l'user qui l'a créé et éventuellement missionId associée. Ce sont typiquement les *bidonvilles* ou *points d'habitat non réglementaire* signalés par les agents sur le terrain. Ils ont un statut d'avancement de traitement. - **Missions** : Une liste de missions fictives, chacune avec id, titre, description, statut (PLANIFIÉE, EN_COURS, TERMINÉE), localisation (commune, préfecture, type=POINT,

geometry coordonnées – point central, ou cela pourrait être étendu à polygon), dates de début/fin, user qui a créé, et la liste des utilisateurs assignés. Par exemple mission_1 est EN_COURS, assignée à user_2. Cela correspond aux missions affichées sur le Dashboard et la carte. - **Changements** : Liste de changements détectés (via satellite/drone). Chaque changement a id, type (EXTENSION_HORIZONTALE, VERTICALE, CONSTRUCTION_NOUVELLE, etc.), description, surface en m^2, dates (dateDetection, dateBefore, dateAfter – pour savoir sur images avant/après), commune/prefecture, statut (DETECTÉ, EN_TRAITEMENT, TRAITÉ), un champ douarId (pour lier ce changement à un douar existant s'il se situe dans un douar connu), chemin des photos avant/après, l'id de l'agent qui a validé la détection, date de création, et la geometry point. Ces changements alimentent la page Changements et la carte. Par ex changement_1 est une extension horizontale détectée le 10/02/2024, surface 45.5m^2, statut DETECTÉ, lié au douar_1, avec photos before_1.jpg et after_1.jpg. - **Actions** : Liste des actions d'intervention réalisées. Chaque action a id, type (DEMOLITION, SIGNALEMENT, REGULARISATION), date, observations, montantAmende, commune, prefecture, douarId lié, missionId lié, userId de l'agent qui l'a faite, un objet user (avec id, name, email – dupliqué pour commodité), date création, createdBy (id user), statut (EN_COURS ou TERMINÉE), et geometry (point). Par ex action_1 est une DEMOLITION terminée le 15/02/2024 par user2 sur douar_1, amende 25000 DH, etc., action_2 un SIGNALEMENT en cours par user1 sur douar_2 sans amende, action_3 une REGULARISATION terminée par user4 sur douar_3 avec amende 5000 DH. On remarque ici que dans `ACTION_TYPES` du code, REGULARISATION n'existe pas – ce désalignement entre mockData et code suggère un besoin d'harmonisation (ils ont mis NON_DEMOLITION à la place, sans le gérer dans les données). - **Fichiers** : Liste des fichiers stockés (liés aux changements ou actions). Par ex fichier_1 et fichier_2 sont les photos avant/après du changement_1, avec chemin, taille, contentType, et lient entityType "CHANGEMENT". fichier_3 est un PV PDF scanné lié à action_1 (démolition Ain Sebaa) avec son chemin. Ces informations ne sont peut-être pas exploitées pleinement dans l'UI actuelle, sauf via la partie PhotoComparer ou PVViewer. - **PVs** : Liste de procès-verbaux. Exemple pv_1 correspond à action_1 (démolition), il a un champ `contenu` qui est un long texte formaté multi-lignes représentant le PV complet (avec Type, Date, Agent, Douar, constatations, décision, sanction, etc.). Ce contenu est ce qui est affiché dans la modale PV du tableau des actions. Le PV a aussi un champ `valide` (booléen), un `urlPDF` (chemin du PDF généré), l'id de l'action correspondante, `redacteurUserId` et `validateurUserId` (ex user_2 a rédigé, user_gouverneur a validé), date de validation, etc.. Ceci simule le cycle de vie d'un PV : un agent crée un brouillon, puis le gouverneur le valide. Un second pv_2 est fourni pour un signalement (action_2), vraisemblablement non validé encore.

Le service `MockApiService` utilise ces données : il stocke en interne des copies (`this.users = [...initialUsers]`, etc.). On note cependant qu'il ne stocke pas explicitement les douars au constructor – peut-être un oubli, car il a bien getDouars() qui utilise `this.douars` mais jamais initialisé. Idem pour actions – il commence avec this.actions = [] au lieu d'y mettre les initialActions du mockData. Cela signifie qu'au chargement initial, `useMock()` va récupérer `actions` vide (sauf si on a ajouté via createAction plus tôt). Pour contourner, dans Actions.js ils prennent `baseActions = apiActions.length ? apiActions : mockData.actions`. Ce genre de fallback est mis en place pour éviter de n'afficher aucune donnée en l'absence d'API réelle. C'est un point à ajuster (intégrer initialActions et initialDouars dans MockApiService).

Les méthodes du service (examinées partiellement plus haut) fournissent toutes les opérations CRUD nécessaires. Par exemple `getMissions()` renvoie la liste complète des missions (après un petit délai simulé), `createMission` push la mission dans le tableau et notifie les abonnées, idem pour update/delete. Il y a des méthodes de recherche `searchMissions(filters)` et `searchChangements(filters)` qui appliquent des filtres server-side si besoin – pour l'instant non utilisées, mais prêtes.

En plus, MockApiService gère `login(email, password)` : il cherche un utilisateur dont l'email correspond et si un password non vide est fourni il retourne un objet { token: "mock_token_xxx", user }. Sinon il throw une erreur. Ce simple check simule l'authentification (pas de vérification de mot de passe réelle).

Enfin, il y a `getStats()` pour compiler les statistiques globales (compte des missions, changements, actions, etc. sur l'ensemble). Ce n'est pas vraiment utilisé dans Dashboard (ils recalculent côté client), mais la page Stats pourrait s'en servir.

Styles et design

L'application s'appuie sur **Ant Design** pour la cohérence visuelle de nombreux composants (tableaux, formulaires, notifications, modals, menus, icônes). Le `ConfigProvider` définit la palette (couleur primaire bleue `#1890ff`, rayons de bord 6px pour un style arrondi léger, etc.). Le thème global est donc homogène.

Les fichiers CSS personnalisés ajoutent par exemple dans `components.css` quelques variables CSS réutilisées (couleurs primaire, succès, warning, etc., textes). Ces variables sont utilisées dans le CSS module ou inline styles pour assurer que, par exemple, la bordure grise `var(--color-gray-200)` reste cohérente sur tout le site. On peut y définir aussi des classes utilitaires globales si nécessaire.

Les CSS modules (ex: `CarteFilters.module.css`, `CarteListe.module.css`, etc.) isolent le style de ces composants. Par exemple, dans CarteFilters.module.css on aura `.filtersContainer`, `.filterGroup`, etc. Ce choix des CSS modules évite les conflits de noms et facilite la maintenance en contexte large.

Le design global est **responsive** : `MainLayout` gère un menu mobile distinct et ajuste tailles de textes dans le header selon `isMobile` (fontSize 16px vs 18px). De plus, la carte adapte l'emplacement des contrôles de zoom. La grille AntD (Row/Col) est utilisée sur le dashboard et d'autres pages pour faire des layouts adaptatifs (par ex. 24 colonnes sur xs, 6 sur lg, etc., pour afficher 4 stats sur une ligne en desktop et 2x2 en mobile). Les composants AntD comme `<ResponsiveGrid>` ne sont pas explicitement utilisés mais la grille l'est.

Au niveau **performance**, le lazy loading des pages permet de ne charger le code d'une page que quand l'utilisateur y navigue (utile car la page Carte avec Leaflet, ou la page NouvelleAction assez lourde, ne sont chargées que si besoin). Le context Notification utilise un reducer pour efficacité, et la plupart des composants utilisent `useMemo` et `useCallback` pour éviter des recalculs ou re-rendus inutiles (par ex. calcul du menuItems dans MainLayout, filtrage dans Dashboard, etc.). Le recours aux bibliothèques externes est maîtrisé : par exemple Leaflet n'est chargé qu'au moment où on va sur la page Carte ou NouvelleAction (via insertion script), évitant d'alourdir le bundle initial. On peut néanmoins signaler quelques améliorations possibles : - **Pré-chargement** : peut-être prévoir de pré-loader la lib Leaflet en arrière-plan après login, pour que la page Carte s'affiche plus vite quand on y va (actuellement il y a un petit délai le temps de charger depuis CDN). - **Pagination/Virtualisation** : les listes et tableaux affichent toutes les données en une page (avec pagination client ou non). S'il y avait des centaines d'actions ou missions, une virtualisation ou une pagination serveur serait à envisager. AntD Table propose la pagination sur données locales (d'ailleurs ils l'activent pageSize=10 sur Actions). Donc ça va pour l'instant.

En termes d'**accessibilité** : on voit des efforts notables (aria-label sur le menu burger, role="navigation" sur le menu principal, role="main" sur le content, aria-current pour l'item de menu actif). La navigation

clavier sur la carte est un plus. On pourrait encore améliorer en s'assurant que chaque icône a un label (ils l'ont fait pour le bouton toggle menu via aria-label dynamique et pour le menu utilisateur). Il faudrait vérifier le contraste des textes sur les fonds de couleur (le menu latéral bleu foncé avec textes blancs est OK, les badges blancs sur fond coloré aussi). L'usage de `<Text type="secondary">` donne un gris clair parfois, veiller à ce que ce soit lisible (AntD maintient un contraste acceptable normalement). Ajouter des raccourcis claviers (par ex. touche pour ouvrir le menu notifications) pourrait être un plus futur.

Fonctionnement global de l'application

En synthèse, le fonctionnement de l'application repose sur les points suivants :

- **Navigation et routing** : Après connexion, l'utilisateur navigue via le menu latéral ou des boutons. Le routeur React Router v6 gère les pages, avec des garde-fous via `ProtectedRoute` pour empêcher l'accès non autorisé. Les URLs reflètent les entités manipulées (dashboard, carte, missions, actions, changements, etc.), ce qui rend l'application bookmarkable et partageable pour certaines vues.

- **Authentification et rôles** : Un système simple sans back-end réel ici, mais qui stocke un token factice et utilise le contexte Auth pour propager l'état de connexion. Les rôles (Agent, DSI, Gouverneur) déterminent ce que l'utilisateur peut voir et faire. Par exemple, un agent d'autorité ne voit pas le menu Utilisateurs ni Statistiques, et s'il tente l'URL directement il sera bloqué par ProtectedRoute. Cette logique est centralisée et donc facile à ajuster (une modification dans AuthContext se répercute partout).

- **Gestion d'état global** : Plutôt que Redux, on utilise des Context natifs. AuthContext gère l'état user et NotificationContext gère l'état notifications (via useReducer). Pour les données métier (missions, actions…), pas de Redux non plus, on utilise un hook custom `useMock()` qui exploite le MockApiService et son mécanisme d'abonnement. Cela fournit une **source de vérité locale** pour les listes d'objets métier. Chaque fois qu'on crée/supprime une entité via le service, le `notifyListeners()` met à jour ces données. Par exemple, quand on appelle `mockApiService.createAction` depuis NouvelleAction, le Dashboard et la page Actions, qui consomment `actions` via `useMock()`, recevront la nouvelle action automatiquement (par le callback passé dans subscribe). Ce choix rend le code relativement **réactif** sans qu'on ait à remonter manuellement les nouveaux objets dans les props ou à faire de nombreux lifts d'état.

- **Appels API simulés** : Le `mockApiService` sert de couche pour accéder/modifier les données. Dans une application réelle, ce serait remplacé par des appels HTTP vers un backend (via fetch/axios). L'architecture actuelle prépare bien cela : on pourrait implémenter un `apiProvider.js` avec des méthodes concrètes et changer `useMock` pour `useApi` assez facilement. En attendant, le mock assure le fonctionnement hors-ligne et de démonstration.

- **Interactions principales** :

- *Créer une mission* : réservé DSI, soit via page NouvelleMission soit potentiellement via un modal. Cela ajoute une mission (MockApiService.notifyListeners fait que la carte et dashboard verront la mission si éligible).
- *Créer un changement* : réservé DSI, via page ChangementCreate. Ajoute un point changement (le dashboard ou carte le montrera).

- *Créer une action* : réservé Agents (et DSI en backup). Via page NouvelleAction (ou ActionForm modale dans certains contextes). Cela ajoute une action et éventuellement un PV brouillon attaché. L'action apparaîtra dans la liste Actions de l'agent et dans le panel du douar/mission concerné. Une notification de type ACTION_COMPLETED peut être envoyée aux DSI/Gouverneur lors de la création d'une action terminée (c'est probablement à appeler dans `handleSubmit` après createAction si l'action type = DEMOLITION ou autre).
- *Valider un PV* : réservé Gouverneur (peut-être DSI aussi). Sur la page PVEditor, le Gouverneur peut marquer *PV validé*. Cela changerait son statut en "VALIDE" via mockApi (qui définit validatedAt et validatedBy). On pourrait alors envoyer une notification du type PV_GENERATED ou un email, etc.
- *Assigner une mission à un agent* : Lors de la création ou mise à jour d'une mission, on peut choisir un ou plusieurs agents assignés (la donnée `assignedUsers`). Quand une mission est assignée, la NotificationContext a prévu un type MISSION_ASSIGNED pour notifier l'agent. En pratique, après createMission, on devrait appeler `notifyMissionAssigned(missionData)` dans NotificationContext pour générer une notif à l'agent concerné. Le code de NotificationContext montre comment c'est fait (il prend mission.agentId et envoie une notif à l'agent).
- *Changement détecté terminé* : Quand un agent finit de traiter un changement (ex: il a démoli la construction, ou constaté la conformité), il enregistre une action correspondante (démolition ou régularisation). Possiblement, cela déclenche une notif de type ACTION_COMPLETED aux supérieurs. De plus, l'administrateur DSI peut mettre à jour le statut du changement en "Traité" via l'interface (non explicitée mais on pourrait imaginer un bouton sur la page Changements pour marquer comme traité).

- *Suppression d'un douar* : Si un douar (bidonville) est complètement éradiqué, on pourrait le supprimer via `deleteDouar`. Le service le permet. L'UI le propose peut-être via un bouton dans CartePanel ou Missions.

- **Notifications en pratique** : Au démarrage, NotificationContext génère une notification fictive selon le rôle. Par exemple un agent aura déjà "Nouvelle mission de surveillance assignée dans votre secteur" non lue. Quand l'agent complète une action (démolition), la page NouvelleAction une fois qu'elle crée l'action devrait appeler `notifyActionCompleted(createdAction)` – or, ce n'est pas visible dans le code, mais c'est prévu dans NotificationContext pour envoyer ACTION_COMPLETED aux DSI/Gouverneur. Idem quand DSI crée un changement ou mission, il devrait appeler notifyChangementDeclared ou notifyMissionCreated pour alerter le Gouverneur. Il serait judicieux d'ajouter ces appels dans le futur afin d'exploiter pleinement le système de notifications déjà en place.

- **Fichiers et uploads** : Le système permet aux agents de joindre des photos et documents. Par exemple, dans NouvelleAction, l'agent peut prendre des photos avant/après si c'est une démolition et ces fichiers seront listés dans l'action créée (champ `photos`). Dans une vraie appli, ils seraient envoyés sur un stockage (cloud ou serveur) et on garderait les URL. Ici, `defaultUploadManager.handleFilesUpload` simule cet upload et fournit une URL (dans `file.tempUrl` ou `file.url`). On voit dans mockData que les images after/before sont référencées pour les changements. Le PV scanné PDF est aussi un fichier. L'interface ne propose pas explicitement de téléverser un PV PDF (on génère du texte via PVEditor), mais possiblement, dans PVEditor ou Utilisateurs on pourrait uploader des documents (ex: annexes). Le DocumentUploader est suffisamment générique pour ça (context PV_ANNEXES par ex. dans `UPLOAD_CONTEXTS`).

- **Sécurité** : L'application étant interne, pas de notions d'inscriptions ou mot de passe fort gérées ici. En prod, il faudrait évidemment sécuriser la connexion, protéger l'API etc. Le design en couches actuelles faciliterait l'ajout d'une Auth via JWT par exemple (il y a déjà un `token` stocké,

même s'il est bidon, qu'on peut remplacer par un vrai JWT et l'envoyer dans les headers d'appels API). Les permissions sont gérées côté front ici, mais dans un contexte réel il faudrait que le backend applique les mêmes restrictions (ne pas renvoyer de données interdites aux requêtes d'un agent, etc.). Étant un outil interne, on peut supposer une confiance plus grande dans l'utilisateur connecté.

Résumé final

Pour conclure, cette application React propose une **architecture modulaire et robuste** adaptée à un outil de gestion professionnelle. Les points forts à retenir sont :

- Une séparation claire des responsabilités : les pages orchestrent la logique métier et l'interface globale, tandis que de nombreux composants réutilisables (formulaires d'action, panneau de carte, filtres, etc.) encapsulent des fonctionnalités précises. Les contextes assurent le partage d'état (session utilisateur, notifications) sans recourir à un état global lourd, ce qui simplifie le code.

- Un système de **permissions fin** est implanté, garantissant que chaque utilisateur ne voit et ne peut agir que sur ce qui le concerne. Ce système est facilement extensible (il suffit d'ajouter un flag dans AuthContext et de l'utiliser dans ProtectedRoute ou dans l'interface conditionnellement). Par exemple, on a pu définir que seul le DSI peut créer ou supprimer des missions, que les agents sont limités à leur commune, etc., et l'UI s'adapte dynamiquement (menu, boutons, routes protégées) en conséquence.

- Une expérience utilisateur riche est fournie : l'**interface cartographique** interactive avec filtres et clusters permet de visualiser les foyers et missions sur le terrain, le tout couplé à des panneaux d'information contextuels (CartePanel) qui évitent de naviguer ailleurs pour voir les détails. La possibilité de sélectionner un emplacement par un simple clic sur la carte ou via GPS facilite grandement la tâche des agents sur le terrain lors de la saisie d'une intervention. Les formulaires intègrent des **retours immédiats** (ex: validation en temps réel via Zod, indicateur de brouillon sauvegardé, etc.) pour guider l'utilisateur. Les notifications temps réel informent chaque acteur des événements importants (nouvelles missions, actions terminées, PV généré…), améliorant la réactivité de la chaîne hiérarchique.

- Du point de vue **code et architecture** : l'utilisation de React Hooks (useEffect, useState, useReducer, useContext, useMemo, useCallback) est bien maîtrisée pour gérer les effets de bord (chargements de données, interactions avec l'API simulée) et optimiser les re-rendus. L'application est segmentée en composants fonctionnels faciles à maintenir et tester isolément. Le recours à Ant Design fournit un style professionnel uniforme et des composants de haute qualité (tableaux, formulaires, etc.), ce qui a accéléré le développement tout en garantissant l'accessibilité et la réactivité du design.

- Côté **performances**, le chargement paresseux des pages lourdes et le fait de n'évaluer certaines choses qu'en fonction des besoins (ex: calculs de stats dans Dashboard sur données filtrées seulement) contribuent à garder l'application fluide. Sur le plan **responsive**, l'interface s'adapte aux écrans plus petits : menu en tiroir, suppression d'éléments non critiques sur mobile (ex: texte du rôle dans le header), usage de la grille pour passer en une colonne sur smartphone, etc.

Les quelques **pistes d'amélioration** identifiées : - Finaliser l'uniformisation entre les données mock et le code (par ex., ajouter le type "REGULARISATION" dans ACTION_TYPES ou ajuster les données pour

utiliser NON_DEMOLITION, initialiser `this.douars` et `this.actions` avec les valeurs de mockData pour éviter d'avoir à faire des fallback dans le front). - Mieux tirer parti du système de notifications en appelant les fonctions du NotificationContext aux endroits stratégiques (après une création de mission, d'action, etc., comme évoqué plus haut) afin que les superviseurs reçoivent immédiatement l'alerte correspondante dans l'UI. - Continuer à renforcer l'accessibilité (par exemple en fournissant un équivalent texte aux badges de couleur seuls, ou en vérifiant les contrastes), même si le gros du travail est déjà fait via AntD et quelques attributs ARIA ajoutés. - Sur le plan performance, on pourrait envisager de ne pas tout stocker en mémoire via useMock dans une version réelle (surtout si le volume de données grossit), mais plutôt d'appeler l'API au fil des besoins et de mettre en cache local ou via un store global léger. Également, l'utilisation de **React Hook Form** avec Zod (comme dans ActionForm) peut être étendue aux autres formulaires pour bénéficier d'une validation plus fluide côté client. - Enfin, l'intégration d'une vraie API backend nécessitera de remplacer progressivement le mock par de vraies requêtes. La structure actuelle le permet sans trop de peine grâce à l'abstraction du service et du contexte.

Malgré ces ajustements mineurs, le projet tel quel est déjà bien structuré pour un usage interne. Il offre aux différents utilisateurs (agents, DSI, gouverneur) une interface centralisée pour collaborer : les agents peuvent déclarer et traiter les infractions avec preuves à l'appui, les DSI supervisent l'ensemble et gèrent les ressources (missions, comptes), le gouverneur peut suivre l'avancement via le tableau de bord et valider les actions officielles (PV). Le code est organisé de façon **claire et pédagogique**, ce qui en facilite la prise en main pour un nouveau développeur rejoignant l'équipe. En somme, l'architecture front-end de cette application est solide et modulable, prête à évoluer avec l'ajout éventuel de nouvelles fonctionnalités (ex: export de données, rapports PDF complets, historique temporel des changements sur la carte, etc.), tout en maintenant un cadre cohérent et maintenable pour le long terme.
