**Government AI Platform - Frontend Developer Guide**

**Table of Contents**

1. [Introduction](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#introduction)
2. [Architecture Overview](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#architecture-overview)
3. [Setup Instructions](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#setup-instructions)
4. [Project Structure](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#project-structure)
5. [Core Technologies](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#core-technologies)
6. [Authentication Flow](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#authentication-flow)
7. [API Integration](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#api-integration)
8. [Component Library](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#component-library)
9. [Internationalization (i18n)](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#internationalization-i18n)
10. [Theme System](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#theme-system)
11. [Security Considerations](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#security-considerations)
12. [Accessibility Implementation](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#accessibility-implementation)
13. [State Management](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#state-management)
14. [Common Development Tasks](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#common-development-tasks)
15. [Troubleshooting](https://claude.ai/chat/58ba3418-313f-41a7-9e54-fb10cf5a4bb0#troubleshooting)

**Introduction**

The Government AI Platform is a secure, accessible web application that provides AI capabilities to government users. It features a chat interface with streaming responses, document management with RAG (Retrieval-Augmented Generation), and comprehensive security measures for handling sensitive government data.

Key features include:

* JWT-based secure authentication
* Real-time streaming chat with markdown support
* Document upload and management
* RAG capabilities for document-augmented AI responses
* Bilingual support (English/French)
* Dark/light mode support
* WCAG 2.1 AA accessibility compliance

**Architecture Overview**

The application is built using Next.js App Router architecture with the following key architectural decisions:

* **App Router Structure**: Pages and layouts organized by locale
* **Server Components**: Used where appropriate for improved performance
* **API Integration Layer**: Centralized API clients for each backend service
* **Authentication Middleware**: Route protection via Next.js middleware
* **Server Actions**: Used for secure form submissions and data mutations
* **Client Components**: Used for interactive UI elements with React hooks

**Data Flow**

1. User interacts with UI components
2. Client components handle state and events
3. API layer communicates with backend services
4. Server components pre-render data where possible
5. Server actions handle secure data mutations
6. Real-time updates via SSE for streaming responses

**Setup Instructions**

**Prerequisites**

* Node.js (v18 or later)
* npm or yarn
* Git

**Installation Steps**

1. Clone the repository:
2. git clone https://github.com/your-org/government-ai-platform.git
3. cd government-ai-platform
4. Install dependencies:
5. npm install
6. # or
7. yarn install
8. Environment setup: Create a .env.local file in the root directory with the following variables:
9. NEXT\_PUBLIC\_API\_URL=http://localhost:8000
10. NEXT\_PUBLIC\_BASE\_URL=http://localhost:3000
11. Start the development server:
12. npm run dev
13. # or
14. yarn dev
15. The application will be available at http://localhost:3000

**Environment Variables**

| **Variable** | **Description** | **Example** |
| --- | --- | --- |
| NEXT\_PUBLIC\_API\_URL | Backend API URL | http://localhost:8000 |
| NEXT\_PUBLIC\_BASE\_URL | Frontend URL for absolute links | http://localhost:3000 |
| NEXTAUTH\_SECRET | Secret for session encryption | random-string |
| NODE\_ENV | Environment (development/production) | development |

**Project Structure**

project-root/

├── app/

│ ├── [locale]/ # Localized routes (en/fr)

│ │ ├── layout.tsx # Root layout with providers

│ │ ├── page.tsx # Home page

│ │ ├── auth/

│ │ │ ├── login/

│ │ │ │ └── page.tsx # Login page

│ │ ├── chat/

│ │ │ └── page.tsx # Chat interface

│ │ ├── documents/

│ │ │ └── page.tsx # Document management

│ │ └── settings/

│ │ └── page.tsx # User settings

│ ├── api/ # API route handlers

│ └── favicon.ico

├── components/

│ ├── auth/ # Auth components

│ ├── chat/ # Chat components

│ ├── documents/ # Document components

│ ├── layout/ # Layout components

│ └── ui/ # Reusable UI components

├── lib/

│ ├── api/ # API integration layer

│ ├── auth/ # Auth utilities

│ ├── i18n/ # Internationalization

│ └── utils/ # Utility functions

├── hooks/ # Custom React hooks

├── context/ # Context providers

├── types/ # TypeScript definitions

├── middleware.ts # Middleware for auth/locale

├── next.config.js # Next.js configuration

├── tailwind.config.js # Tailwind CSS config

├── tsconfig.json # TypeScript config

└── package.json # Dependencies

**Key Directories Explained**

* **app/[locale]**: Contains all pages and layouts with locale prefix routes
* **components**: Reusable React components organized by feature
* **lib**: Non-component code including API clients and utilities
* **hooks**: Custom React hooks for shared logic
* **context**: React context providers for global state
* **types**: TypeScript type definitions
* **middleware.ts**: Next.js middleware for auth and localization

**Core Technologies**

**Frontend Framework**

* **Next.js 14**: React framework with App Router architecture
* **React 18**: UI component library
* **TypeScript**: Static typing for improved developer experience

**Styling**

* **Tailwind CSS**: Utility-first CSS framework
* **clsx & tailwind-merge**: For conditional class name handling
* **CSS Modules**: For component-scoped styling

**Data Fetching**

* **Fetch API**: Native browser API for data fetching
* **Server Components**: For server-side data fetching
* **Server Actions**: For secure form submissions

**Authentication**

* **JWT**: JSON Web Tokens for authentication
* **HTTP-only Cookies**: For secure token storage

**Internationalization**

* **React Intl**: For internationalization
* **Next.js Middleware**: For locale detection and routing

**UI Components**

* **Headless UI**: Unstyled, accessible UI components
* **React Hook Form**: For form handling
* **Zod**: For form validation

**Other Libraries**

* **date-fns**: For date formatting and manipulation
* **React Markdown**: For markdown rendering
* **Prism.js**: For code syntax highlighting

**Authentication Flow**

The application uses JWT-based authentication with the following flow:

1. **Login**: User submits credentials to /api/v1/auth/login
2. **Token Storage**: JWT is stored in HTTP-only, secure cookies
3. **Token Validation**: Middleware validates token on protected routes
4. **Token Refresh**: Automatic refresh when token expires
5. **Logout**: Token removal and redirect to login

**Implementation Details**

* Authentication state is managed via React Context
* The useAuth hook provides authentication state to components
* Protected routes are guarded by middleware
* Token refresh is handled automatically by the API layer

**API Integration**

The application uses a modular API client architecture:

**Base API Client**

Located in lib/api/base.ts, the base client handles:

* Token inclusion in requests
* Error handling
* Response parsing
* Token refresh on 401 errors

**Feature-specific API Clients**

* **AuthApi**: Authentication operations (lib/api/auth.ts)
* **ChatApi**: Chat and conversation operations (lib/api/chat.ts)
* **DocumentsApi**: Document management operations (lib/api/documents.ts)

Example usage:

// Server component

const documents = await documentsApi.getDocuments({ limit: 10 });

// Client component with React hooks

const [documents, setDocuments] = useState([]);

useEffect(() => {

const fetchData = async () => {

const result = await documentsApi.getDocuments({ limit: 10 });

setDocuments(result.documents);

};

fetchData();

}, []);

**Component Library**

The application uses a custom component library built on Tailwind CSS:

**Core UI Components**

* **Button**: Multi-variant button component (components/ui/button.tsx)
* **Input**: Form input component (components/ui/input.tsx)
* **Textarea**: Multiline text input (components/ui/textarea.tsx)
* **Dropdown**: Selection component (components/ui/dropdown.tsx)
* **Modal**: Dialog component (components/ui/modal.tsx)
* **Toast**: Notification component (components/ui/toast.tsx)

**Feature Components**

* **ChatWindow**: Chat interface (components/chat/chat-window.tsx)
* **MessageList**: Message rendering (components/chat/message-list.tsx)
* **DocumentList**: Document management (components/documents/document-list.tsx)
* **DocumentUpload**: File upload interface (components/documents/document-upload.tsx)

**Layout Components**

* **Header**: Application header with navigation (components/layout/header.tsx)
* **Footer**: Application footer (components/layout/footer.tsx)
* **Sidebar**: Navigation sidebar (components/layout/sidebar.tsx)

**Internationalization (i18n)**

The application supports English and French with the following implementation:

**Locale Routing**

* Routes are prefixed with locale code (/en/chat, /fr/documents)
* Default locale is set to English (en)
* Middleware detects and redirects to appropriate locale

**Translation System**

* Translation keys are stored in lib/i18n/translations.ts
* Server-side translations via getTranslations function
* Client-side translations via useTranslation hook

**Usage Examples**

Server component:

const t = await getTranslations(locale);

return <h1>{t("home.welcome")}</h1>;

Client component:

const { t, locale, changeLocale } = useTranslation();

return (

<>

<h1>{t("home.welcome")}</h1>

<button onClick={() => changeLocale("fr")}>Français</button>

</>

);

**Theme System**

The application supports light and dark modes:

**Theme Provider**

* Context provider in context/theme-provider.tsx
* Theme state (light/dark/system) with localStorage persistence
* Media query detection for system preference

**Usage**

const { theme, setTheme } = useTheme();

// Toggle theme

<button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>

Toggle theme

</button>

**CSS Implementation**

* Tailwind dark mode via dark: prefix classes
* CSS variables for custom theming
* System preference detection via prefers-color-scheme

**Security Considerations**

The application implements several security measures:

**Authentication Security**

* HTTP-only, secure cookies for token storage
* JWT with short expiration and refresh token rotation
* CSRF protection via SameSite cookie attributes
* Rate limiting on authentication endpoints

**Content Security**

* Content Security Policy (CSP) headers
* XSS protection via React's built-in protections
* Input sanitization for all user inputs
* Proper output encoding for dynamic content

**Data Protection**

* HTTPS enforcement in production
* Secure handling of Protected B government data
* Input validation for all form data
* Secure document upload validation

**Accessibility Implementation**

The application follows WCAG 2.1 AA standards:

**Key Accessibility Features**

* Semantic HTML structure
* Keyboard navigation support
* ARIA attributes where appropriate
* Focus management
* Screen reader compatibility
* Sufficient color contrast
* Reduced motion support
* Responsive design for all devices

**Testing Tools**

* Axe DevTools for automated accessibility testing
* Lighthouse for accessibility audits
* Screen reader testing with NVDA and VoiceOver

**State Management**

The application uses a combination of state management approaches:

**Local Component State**

* React useState and useReducer for component-specific state
* React useRef for non-reactive values

**Context API**

* AuthContext: User authentication state
* ThemeContext: Theme preferences
* NotificationContext: Toast notifications

**Server State**

* Server Components for initial data loading
* React Query for client-side data fetching and caching

**Common Development Tasks**

**Adding a New Page**

1. Create a new file in the appropriate directory under app/[locale]/
2. Export a React component as the default export
3. Add any necessary data fetching
4. Update navigation if needed

Example:

// app/[locale]/examples/page.tsx

export default async function ExamplesPage({

params: { locale },

}: {

params: { locale: string };

}) {

const t = await getTranslations(locale);

return (

<div className="container mx-auto py-8">

<h1 className="text-2xl font-bold">{t("examples.title")}</h1>

{/\* Page content \*/}

</div>

);

}

**Adding a New Component**

1. Create a new file in the appropriate directory under components/
2. Export the component with appropriate TypeScript interfaces
3. Use the component in pages or other components

Example:

// components/ui/card.tsx

interface CardProps {

title: string;

children: React.ReactNode;

className?: string;

}

export function Card({ title, children, className }: CardProps) {

return (

<div className={`bg-white dark:bg-gray-800 rounded-lg p-6 shadow-sm ${className}`}>

<h3 className="text-lg font-medium mb-4">{title}</h3>

{children}

</div>

);

}

**Adding New API Integration**

1. Create a new file in lib/api/ if needed
2. Define interfaces for request/response data
3. Implement API methods using the base API client

Example:

// lib/api/notifications.ts

import { apiClient } from './base';

import type { Notification } from '@/types/notifications';

export class NotificationsApi {

async getNotifications(): Promise<Notification[]> {

return await apiClient.get<Notification[]>('/api/v1/notifications');

}

async markAsRead(id: string): Promise<void> {

await apiClient.post<void>(`/api/v1/notifications/${id}/read`, {});

}

}

export const notificationsApi = new NotificationsApi();

**Adding New Translations**

1. Add new translation keys to lib/i18n/translations.ts
2. Add translations for all supported languages
3. Use the translations in components with t() function

Example:

// lib/i18n/translations.ts

export const translations = {

en: {

// ...existing translations

newFeature: {

title: "New Feature",

description: "This is a new feature in the application."

}

},

fr: {

// ...existing translations

newFeature: {

title: "Nouvelle Fonctionnalité",

description: "C'est une nouvelle fonctionnalité dans l'application."

}

}

};

**Troubleshooting**

**Common Issues and Solutions**

**Authentication Issues**

**Issue**: User is unexpectedly logged out or redirected to login **Solution**: Check cookie settings and expiration, ensure CSP isn't blocking cookies

**API Connection Problems**

**Issue**: API requests fail with network errors **Solution**: Verify API URL in environment variables and network connectivity

**Styling Inconsistencies**

**Issue**: Components look different in development vs production **Solution**: Check for CSS purging issues in Tailwind configuration

**Build Errors**

**Issue**: TypeScript errors during build **Solution**: Run tsc --noEmit to check for type errors and fix them

**Debugging Tips**

1. Use React DevTools for component inspection
2. Check browser console for errors
3. Use Network tab to inspect API requests
4. Enable source maps for better debugging
5. Use console.log strategically (remove before committing)

**Getting Help**

* Open an issue in the project repository
* Check existing documentation in the team wiki
* Consult with the DevOps team for deployment-related issues
* Refer to Next.js documentation for framework-specific questions