Frontend Authentication Middleware Implementation

Introduction

This technical document outlines the implementation of the authentication middleware in the AKA Studio marketing automation platform. The system transitions from a client-side authentication approach using the AuthProvider component to a more robust middleware solution. This document covers the implementation details, benefits, and integration with backend cookie-based authentication.

Background

Previously, the application relied on a React context-based authentication system (`AuthProvider`) which had several limitations:

1. Authentication state could only be verified after the component tree mounted
2. Unprotected access to pages during initial load
3. Unnecessary rendering of unauthorized content before redirects
4. Potential flash of unauthenticated content (FOUC)

Middleware Implementation

Core Architecture

The Next.js middleware architecture provides several benefits:

- **Server-Side Execution**: Runs before any page or API route is rendered
- **Universal Coverage**: Applied to all routes without requiring explicit inclusion in components
- **Performance**: Minimizes unnecessary component rendering
- **Security**: Prevents unauthorized access at the routing level

Implementation Details

```
// src/middleware.ts
import { NextResponse } from "next/server";
import type { NextRequest } from "next/server";

// Define protected routes pattern
const PROTECTED_ROUTES = /^\/(dashboard|settings|posts|promotions)/;

// Define authentication check routes
const AUTH_ROUTES = /^\/(login|register)/;

// Define routes that require a business to be set up
const BUSINESS_REQUIRED_ROUTES =
  /^\/(posts|promotions|settings(\/(?!general).+))/;

export function middleware(request: NextRequest) {
  const { pathname } = request.nextUrl;

  // Check if we're on a protected route
  const isProtectedRoute = PROTECTED_ROUTES.test(pathname);

  // Check if we're on an auth route (login/register)
  const isAuthRoute = AUTH_ROUTES.test(pathname);

  // Check if the route requires business setup to be completed
  const isBusinessRequiredRoute = BUSINESS_REQUIRED_ROUTES.test(pathname);

  // Get authentication token from cookies
  const token = request.cookies.get("access_token")?.value;

  // Get business ID from cookies
  const businessId = request.cookies.get("business_id")?.value;

  // If trying to access protected route without auth token, redirect to login
  if (isProtectedRoute && !token) {
    return NextResponse.redirect(new URL("/login", request.url));
  }

  // If authenticated user tries to access auth routes, redirect to dashboard
  if (isAuthRoute && token) {
    return NextResponse.redirect(new URL("/dashboard", request.url));
  }

  // If trying to access business required route without business id, redirect to
settings
  if (isBusinessRequiredRoute && token && !businessId) {
    return NextResponse.redirect(new URL("/settings/general", request.url));
  }

  // Continue with the request for all other cases
  return NextResponse.next();
}

// Configure middleware to run on specific paths
export const config = {
  matcher: [
    // Match all routes except static files, api routes, etc.
    "/((?!api|_next/static|_next/image|favicon.ico).*)",
  ],
};
```

Integration with Backend Cookies

The backend sets two critical cookies that the middleware relies on:

1. `access_token`: Session authentication token
2. `business_id`: Identifier for the user's business

These cookies are set with the following properties:

- `httpOnly: true` (prevents JavaScript access)
- `secure: true` (requires HTTPS)
- `samesite: "None"` (allows cross-site requests)
- `path: "/"` (accessible across the whole domain)

Backend Cookie Management Code Sample

```
# backend/businesses/views.py (snippet)
def _update_business(self, request, partial=False):
    # Create business and set business_id cookie
    Serializer = BusinessSerializer(data=request.data, context={'request':
request})
    if serializer.is_valid():
        business = serializer.save(owner=request.user)
        response = Response(serializer.data, status=status.HTTP_201_CREATED)

        # Set business_id cookie
        response.set_cookie(
            key="business_id",
            value=str(business.id),
            path="/",
            httponly=True,
            secure=True,
            samesite="None",
        )
        return response
```

```python
# backend/users/views.py (snippet)
def login(self, request):
    # Authentication logic...

    # Set access_token cookie
    response = Response({
        "message": "Login successful",
        "refresh": str(tokens)
    }, status=status.HTTP_201_CREATED)

    response.set_cookie(
        key=settings.SIMPLE_JWT["AUTH_COOKIE"], # "access_token"
        value=str(tokens.access_token),
        httponly=settings.SIMPLE_JWT["AUTH_COOKIE_HTTP_ONLY"], # True
        secure=settings.SIMPLE_JWT["AUTH_COOKIE_SECURE"], # True
        samesite=settings.SIMPLE_JWT["AUTH_COOKIE_SAMESITE"], # None
        expires=60 * 60 * 24, # Valid for 1 day
    )
    return response
```

Comparison with Previous AuthProvider Implementation

Limitations of Previous Approach

1. **Late Authentication Check**: Authentication status is only determined after component mounting and API call completion
2. **Client-Side Only**: Security checks happen only in the browser
3. **Flash of Unauthorized Content**: Users may briefly see protected content before redirect
4. **Redundant Rendering**: Pages render before determining authentication status

Benefits of the New Implementation

1. **Early Interception**: Authentication is verified before any component renders
2. **Enhanced Security**: Unauthorized users never receive protected page content
3. **Improved UX**: No flashes of unauthorized content or unnecessary redirects
4. **Reduced Client-side Logic**: Simplified component code without authentication checks
5. **Performance**: Fewer wasted renders and network requests

Technical Workflow

1. User requests a route (e.g., `/dashboard`, `/settings`, `/login`, etc.)
2. Middleware intercepts the request before any component is rendered
3. Middleware checks for the `access_token` (authentication) and `business_id` (business setup) cookies
4. If the route is protected and `access_token` is missing → Redirect to `/login`
5. If the route is `/login` or `/register` and `access_token` exists → Redirect to `/dashboard`
6. If the route requires business setup and `business_id` is missing → Redirect to `/settings/general`
7. Otherwise → Allow the request to proceed to the intended route

Backward Compatibility

The AuthProvider component remains in the codebase but with a reduced role:

- Provides the user data and authentication state to components that need it
- Handles authentication operations (login, logout, register)
- No longer responsible for route protection or redirects

Conclusion

The transition from a client-side AuthProvider to a server-side middleware approach significantly improves the authentication system's security, performance, and user experience. The middleware implementation provides earlier interception of unauthorized requests, eliminates unnecessary rendering, and works seamlessly with the backend's cookie-based authentication system.

This architecture follows modern best practices for web application security by leveraging HTTP-only cookies for authentication tokens and implementing route protection at the server level before any client-side code executes.