Authentication Implementation Guide for Next.js with Django Backend

1. Project Requirements and Authentication Goals

Our project implements a web application with Next.js frontend and Django backend, with specific authentication requirements:

1. **Secure token management**: Must protect authentication credentials from XSS and CSRF attacks
2. **Smooth user experience**: Authentication state should persist across page refreshes without UI flickering
3. **Future extensibility**: The system must support planned features:
    - Social login integration (Google, Facebook, Apple)
    - Passkey authentication (biometric/WebAuthn)
    - Two-factor authentication (2FA)

This document explains our authentication implementation decisions, how they satisfy our requirements, and how we solved common authentication challenges.

2. Authentication Implementation

Our application uses a secure JWT-based authentication system with HTTP-only cookies, integrating a Next.js frontend with a Django backend. This section explains the core architecture and security considerations.

2.1. JWT with HTTP-only Cookies Architecture

The authentication system follows these key principles:

1. **Secure Token Storage**: JWT tokens are stored in HTTP-only cookies, not in localStorage or sessionStorage
2. **Backend Token Validation**: Django backend validates tokens and manages session state
3. **CSRF Protection**: Cross-Site Request Forgery protection is implemented on the backend
4. **Custom Authentication Class**: A custom JWT authentication class in Django processes the HTTP-only cookies

This approach provides several security benefits:

- **XSS Protection**: HTTP-only cookies cannot be accessed by JavaScript, protecting against Cross-Site Scripting attacks
- **CSRF Mitigation**: Proper CSRF tokens prevent Cross-Site Request Forgery
- **Token Security**: Authentication tokens never expose to client-side JavaScript

The authentication flow works as follows:

1. User submits credentials (email/password)
2. Backend validates credentials and issues a JWT token
3. Token is set in an HTTP-only cookie with secure attributes

4. Subsequent requests automatically include the cookie
5. Backend extracts and validates the token from the cookie

3. Authentication Approaches Comparison

We evaluated three main approaches for implementing authentication:

Approach	Description	Pros	Cons
1. Django REST Framework with JWT	Using DRF's built-in JWT functionality with `djangorestframework-simplejwt`	Less custom code, established patterns, well-documented	Does not support HTTP-only cookie authentication by default
2. Custom JWT Implementation	Custom authentication class and handlers	Complete control over auth flow, tailored to specific needs	More code to maintain, potential for security oversights
3. Next Auth.js	Frontend-focused authentication library	Built-in social login, passkeys, session management	More complex integration with Django backend

3.1. Current Industry Practice

The industry is currently in transition regarding authentication best practices:

1. **Traditional Backend-Centric Authentication (Approaches 1 & 2)**:
    - Security-critical logic resides on the backend
    - Frontend primarily stores and forwards tokens
    - Still dominant in projects with full-stack backends (Django, Rails, Spring)
    - Preferred for applications with complex authorization requirements
2. **Frontend-Centric Authentication (Approach 3)**:
    - Gaining popularity, especially in serverless and JAMstack architectures
    - Frontend handles OAuth flows, session management, etc.
    - Backend focuses only on token validation and resource protection
    - Well-suited for frontend-only or microservices architectures

3.2. Our Choice: Approach 2 (Custom JWT Implementation)

We chose the custom JWT implementation approach because:

1. **Security requirements**: HTTP-only cookies provide better protection against XSS attacks compared to localStorage
2. **Integration with existing Django backend**: Our backend already implements specific authorization logic
3. **Control over the authentication flow**: We needed to customize token handling for our specific requirements

> **Why Approach 1 (JWTAuthentication) is Vulnerable to XSS Attacks**
>
> 1. Default JWTAuthentication only retrieves tokens from the `Authorization` header.
>
> - The frontend must store and manually send the token in API requests.
>
> 2. Token storage in frontend introduces security risks.
>
> - To include `Authorization: Bearer <token>`, the frontend needs to store the token in: LocalStorage, SessionStorage, or Memory.
> - LocalStorage and SessionStorage are vulnerable to XSS attacks since JavaScript can access them.
> - While storing tokens in memory reduces the XSS risk, it causes authentication loss on page refresh, negatively affecting user experience.

4. Solving the Page Refresh UI Flicker Issue

4.1. Understanding the Problem

When using SWR for authentication state management, page refreshes introduce a challenging issue:

1. SWR's cache is stored in memory and is completely cleared on page refresh
2. On refresh, the authentication state temporarily reverts to `undefined` while SWR fetches the user data
3. This causes protected routes to briefly redirect to the login page before returning to the original page
4. The UI may flicker between authenticated and unauthenticated states

This happens because:

```
// Initial SWR setup
const { data: user, error, mutate } = useFetchData<User | null>
(USERS_API.ME, {
  revalidateIfStale: false, // Don't refetch on mount if data exists
  revalidateOnFocus: false, // Don't refetch when tab regains focus
});
```

During a refresh, `user` starts as `undefined` until the API call completes, regardless of SWR settings.

4.2. Frontend Authentication State Management

On the frontend, we maintain a client-side authentication state using a discriminated union type:

```
type AuthState =
  | { status: "initializing" } // During initial auth check
  | { status: "authenticated", user: User } // User is logged in
  | { status: "unauthenticated" }; // User is not logged in
```

This state is synchronized with the backend authentication status through an API call to `/api/users/me` and is used for UI rendering decisions while keeping the actual JWT token secure in HTTP-only cookies.

4.3. Solution: Using Discriminated Union for Auth State

Our solution uses an explicit authentication state model with a discriminated union type:

```
// Authentication provider with explicit state tracking
const [authState, setAuthState] = useState<AuthState>({ status:
"initializing" });

// Update auth state only after data or error is received
useEffect(() => {
  if (user !== undefined || error) {
    if (error || !user) setAuthState({ status: "unauthenticated" });
    else setAuthState({ status: "authenticated", user });
  }
}, [user, error]);
```

This approach has several advantages:

1. **Type safety**: TypeScript ensures we handle all possible states
2. **Explicit state representation**: The current state is clearly identifiable
3. **No boolean flag variables**: Avoids multiple boolean flags that can lead to inconsistent states

In protected components, we make UI decisions based on the auth state:

```
// Protected layout example
if (authState.status === "initializing") {
  return children; // Keep showing current UI during initialization
}

if (authState.status === "unauthenticated") {
  return null; // Show nothing while redirecting
}
```

This pattern completely eliminates UI flickering during page refreshes by ensuring we only make authentication-dependent UI decisions after the authentication check is complete.

5. Best Practices for Auth-Dependent Components

Different components need different strategies for handling authentication state, especially during initialization. Here are patterns for common scenarios:

5.1. Protected Routes/Layouts

For components that protect entire routes:

```
function ProtectedLayout({ children }) {
  const { authState } = useAuth();
  const router = useRouter();

  // Redirect unauthenticated users after initialization
  useEffect(() => {
    if (authState.status === "unauthenticated") {
      router.push("/login");
    }
  }, [authState, router]);

  // During initialization, preserve the current UI
  if (authState.status === "initializing") {
    return children;
  }

  // After initialization, show nothing during redirect
  if (authState.status === "unauthenticated") {
    return null;
  }

  // User is authenticated, show protected content
  return <div>{children}</div>;
}
```

5.2. Navigation Components

For navigation bars that adapt to auth state:

```
function AuthButtons() {
  const { authState, logout } = useAuth();

  // Show invisible placeholder during initialization
  if (authState.status === "initializing") {
    return <div className="opacity-0">Placeholder</div>;
  }

  // Show appropriate buttons based on auth state
  if (authState.status === "authenticated") {
    return (
      <div className="flex">
        <Link href="/dashboard">Dashboard</Link>
        <button onClick={logout}>Logout</button>
      </div>
    );
  }

  return (
    <div className="flex">
      <Link href="/login">Login</Link>
      <Link href="/register">Register</Link>
    </div>
  );
}
```

5.3. Landing Page

For the main page that redirects authenticated users:

```
function HomePage() {
  const { authState } = useAuth();
  const router = useRouter();

  // Redirect authenticated users to dashboard
  useEffect(() => {
    if (authState.status === "authenticated") {
      router.push("/dashboard");
    }
  }, [authState, router]);

  // Show landing page for unauthenticated and initializing users
  if (authState.status !== "authenticated") {
    return <LandingPageContent />;
  }

  // Show nothing during redirect
  return null;
}
```

The key principles across all components are:

1. Always check auth state before making UI decisions
2. Handle initialization state separately from authenticated/unauthenticated states
3. Maintain UI stability during initialization to prevent flickering
4. Use appropriate redirects based on auth state after initialization is complete

6. Future Extensions and Considerations

The current implementation provides a solid foundation for authentication in our application. Below are recommendations for future enhancements and improvements:

6.1. Recommended Code Structure Improvements

Before implementing additional authentication methods, we should refactor the current JWT token generation logic:

1. **Current implementation**: JWT token generation occurs in `TraditionalLoginSerializer`
2. **Recommended improvement**: Move JWT token generation to the `LoginView`

This refactoring would:

- Centralize token generation in one place (`LoginView`)
- Make authentication serializers simpler by focusing only on user validation
- Reduce code duplication across different authentication serializers
- Ensure a clear separation between authentication logic (serializers) and token management (views)
- Make adding new authentication methods simpler and less error-prone

With this improvement:

- Each serializer will only return a validated user.
- LoginView will be responsible for generating and setting JWT tokens.
- This approach ensures flexibility and security while preventing token issuance before full authentication is complete.
- If 2FA is required, LoginView will check for it and return a "2FA required" response instead of issuing tokens. (See section 6.4 for details.)

6.2. Adding Social Login

Our LoginView already supports multiple authentication methods through different serializers. To add social login:

1. **Integrate django-allauth**: Add django-allauth to handle OAuth flows with social providers
2. **Implement SocialLoginSerializer**:

   ```
   class SocialLoginSerializer(serializers.Serializer):
       provider = serializers.CharField()
       access_token = serializers.CharField()

       def validate(self, data):
           # Use django-allauth to validate the social token
           # Upon successful validation, get or create a user
           # Return authenticated user object
   ```

3. **Maintain single endpoint**: Keep using the same `/api/users/login/` endpoint with `method: "social"` parameter
4. **Unified JWT handling**: LoginView handles JWT token generation for all authentication methods

6.3. Adding Passkeys/WebAuthn

For passkey authentication:

- Add `django-webauthn` to the backend
- Implement `PasskeyLoginSerializer` that validates WebAuthn credentials
- Add registration and validation endpoints for passkeys
- Return authenticated user from serializer, letting LoginView handle token generation

6.4. Two-Factor Authentication

For 2FA support:

- Add `django-two-factor-auth` to the backend
- Implement `TwoFactorVerificationSerializer` for validating OTP codes
- Add endpoints for 2FA setup and verification
- Extend the authentication flow to check for and require 2FA when enabled

7. Conclusion

The implemented authentication system balances security, user experience, and extensibility. By using HTTP-only cookies with JWT tokens and implementing proper initialization state handling, we've created a secure authentication system that provides a smooth user experience even during page refreshes.

While there are alternative approaches like Next Auth.js that offer more built-in features, our current implementation offers good integration with the Django backend while maintaining the flexibility to add more authentication features in the future.