Web Authentication Documentation

Overview

The SSVproff web application implements a complete authentication system using Next.js, React Context, and JWT tokens. It provides a seamless user experience with protected routes and persistent authentication state.

Architecture

Components

1. **API Client** (`src/lib/api.ts`)
   - Type-safe API calls
   - Automatic token injection
   - Error handling

2. **Auth Utilities** (`src/lib/auth.ts`)
   - Token storage (localStorage)
   - Authentication state checks

3. **Auth Context** (`src/contexts/AuthContext.tsx`)
   - Global authentication state
   - Login/logout/register functions
   - User information management

4. **Protected Route** (`src/components/ProtectedRoute.tsx`)
   - Route protection wrapper
   - Automatic redirect to login
   - Loading states

5. **Pages**
   - `/` - Landing page
   - `/login` - Login page
   - `/register` - Registration page
   - `/dashboard` - Main authenticated page with tasks
   - `/profile` - User profile page

Setup Instructions

1. Install Dependencies

```
cd web
npm install
```

2. Configure Environment

```
cp .env.local.example .env.local
```

Update `.env.local`:

```
NEXT_PUBLIC_API_URL=http://localhost:8000/api/v1
```

3. Run Development Server

```
npm run dev
```

The application will be available at http://localhost:3000

Usage

User Flow

1. **Registration**
   - Navigate to `/register`
   - Enter email, username, and password
   - Auto-login after successful registration
   - Redirect to dashboard

2. **Login**
   - Navigate to `/login`
   - Enter email and password
   - Receive JWT tokens
   - Redirect to dashboard

3. **Authenticated Access**
   - Access protected pages (dashboard, profile)
   - Tokens stored in localStorage
   - Automatic token injection in API calls

4. **Logout**
   - Click logout button
   - Tokens cleared from localStorage
   - Redirect to login page

Authentication Context

The `AuthContext` provides:

```
interface AuthContextType {
  user: User | null;           // Current user or null
  loading: boolean;            // Loading state
  error: string | null;        // Error message
  login: (data: LoginData) => Promise<void>;
  register: (data: RegisterData) => Promise<void>;
  logout: () => void;
}
```

Using the Auth Context

```
import { useAuth } from '../contexts/AuthContext';

function MyComponent() {
  const { user, login, logout } = useAuth();

  if (!user) {
    return <div>Please log in</div>;
  }

  return (
    <div>
      <p>Welcome, {user.username}!</p>
      <button onClick={logout}>Logout</button>
    </div>
  );
}
```

Protected Routes

Wrap any page that requires authentication:

```
import { ProtectedRoute } from '../components/ProtectedRoute';

export default function MyProtectedPage() {
  return (
    <ProtectedRoute>
      <div>This content is only visible to authenticated users</div>
    </ProtectedRoute>
  );
}
```

Features:
- Automatic redirect to `/login` if not authenticated
- Loading state while checking authentication
- Seamless user experience

API Client

The API client (`src/lib/api.ts`) provides typed functions for all API endpoints:

Authentication

```
import { login, register, getCurrentUser } from '../lib/api';

// Login
const tokens = await login({
  email: 'user@example.com',
  password: 'password123'
});

// Register
const user = await register({
  email: 'user@example.com',
  username: 'johndoe',
  password: 'password123'
});

// Get current user
const user = await getCurrentUser();
```

Tasks

```
import { getTasks, createTask, updateTask, deleteTask } from '../lib/api';

// Get all tasks
const tasks = await getTasks({ limit: 10, completed: false });

// Create task
const newTask = await createTask({
  title: 'Complete project',
  description: 'Finish implementation'
});

// Update task
const updated = await updateTask(taskId, {
  is_completed: true
});

// Delete task
await deleteTask(taskId);
```

Error Handling

The API client throws `APIError` for failed requests:

```
import { APIError } from '../lib/api';

try {
  await login({ email, password });
} catch (error) {
  if (error instanceof APIError) {
    console.error(`Error ${error.status}: ${error.message}`);
    // error.details contains additional information
  }
}
```

Security Considerations

Token Storage

Currently, tokens are stored in `localStorage`:

Pros:
- Easy to implement
- Persists across page reloads

Cons:
- Vulnerable to XSS attacks

Production Recommendation:
For production, consider using httpOnly cookies:
1. Store tokens in httpOnly cookies on the backend
2. Remove localStorage usage
3. Cookies are automatically sent with requests
4. Protected from XSS attacks

CORS Configuration

Ensure the API allows requests from your frontend:

```
# In API configuration
BACKEND_CORS_ORIGINS=["http://localhost:3000", "https://yourdomain.com"]
```

HTTPS

Always use HTTPS in production to protect tokens in transit.

Customization

Styling

The application uses CSS Modules for styling. Customize by editing:
- `src/styles/globals.css` - Global styles
- `src/styles/Auth.module.css` - Login/register pages
- `src/styles/Dashboard.module.css` - Dashboard page
- `src/styles/Profile.module.css` - Profile page

Adding New Protected Pages

1. Create the page component:

```
// src/pages/my-page.tsx
import { ProtectedRoute } from '../components/ProtectedRoute';

export default function MyPage() {
  return (
    <ProtectedRoute>
      {/* Your content here */}
    </ProtectedRoute>
  );
}
```

1. Add navigation links in other components:

```
<Link href="/my-page">My Page</Link>
```

Extending the API Client

Add new API functions in `src/lib/api.ts`:

```
export interface MyResource {
  id: string;
  name: string;
  // ... other fields
}

export async function getMyResources(): Promise<MyResource[]> {
  return apiRequest<MyResource[]>('/my-resources');
}

export async function createMyResource(data: any): Promise<MyResource> {
  return apiRequest<MyResource>('/my-resources/', {
    method: 'POST',
    body: JSON.stringify(data),
  });
}
```

Testing

Run Tests

```
npm test
```

Test Structure

```
__tests__/
  pages/
    index.test.tsx
  setup.test.ts
```

Deployment

Build for Production

```
npm run build
```

Environment Variables

For production, set:

```
NEXT_PUBLIC_API_URL=https://api.yourdomain.com/api/v1
```

Deploy to Vercel

```
npm install -g vercel
vercel
```

Deploy to Netlify

```
npm run build
# Upload `out/` directory to Netlify
```

Troubleshooting

“Cannot find module” Errors

Ensure TypeScript paths are configured in `tsconfig.json`:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/*": ["src/*"]
    }
  }
}
```

API Connection Issues

1. Check API is running: `curl http://localhost:8000/health`
2. Verify CORS configuration
3. Check `NEXT_PUBLIC_API_URL` in `.env.local`

Authentication Not Persisting

1. Check browser localStorage: DevTools > Application > Local Storage
2. Verify tokens are being stored after login
3. Check console for errors

Additional Resources

- Next.js Documentation (https://nextjs.org/docs)
- React Context API (https://react.dev/reference/react/useContext)
- TypeScript Documentation (https://www.typescriptlang.org/docs/)