Architecture Documentation

Version: 0.1.0
Generated: 7/22/2025

Architecture Overview

This document provides a comprehensive description of the Vydeo application architecture. The application is built as a modern React single-page application (SPA) with a focus on real-time communication, video content management, and social features.

High-Level Architecture

The Vydeo application follows a client-side architecture built with React 18, using Redux for global state management and Context API for theme and UI state. The application communicates with multiple backend services via RESTful APIs and uses WebSockets for real-time features including chat, notifications, and content updates.

The application architecture consists of the following major layers:

- Presentation Layer: React components with Material UI, organized by feature
- State Management Layer: Redux store for global state and Context API for theme and UI state
- Service Layer: API clients, WebSocket communication, and utility services
- Persistence Layer: LocalForage and browser storage for offline data access

Data Flow Diagram:

System Context

The Vydeo application interacts with several external systems:

- Java API Service (https://apis.vydeo.xyz/java): Primary RESTful backend for core functionality
- Python API Service (https://apis.vydeo.xyz/py): Specialized service for data processing

```
    const token = localStorage.getItem("token");
    if (token) {
      config.headers.token = token;
    }
    return config;
  }
);
```

Core Components

Frontend Architecture

The frontend application is organized into feature-based components with the following structure:

- src/ - Source code directory
- % % % components/ - React components organized by feature
- % % % % AccountIssue/ - Authentication related components
- % % % % % Login/ - Login functionality
- % % % % % SignUp/ - Registration functionality
- % % % % % Forget/ - Password recovery
- % % % % Contents/ - Main content components
- % % % % % Videos/ - Video browsing and playback
- % % % % % Chat/ - Messaging functionality
- % % % % % FriendList/ - Friend management
- % % % % % VideoList/ - Video listings and recommendations
- % % % % % Settings/ - User preferences
- % % % % Header/ - Navigation and search components
- % % % % Errors/ - Error handling components
- % % % % redux/ - Redux state management
- % % % Providers/ - Context providers
- % % % % NotificationProvider.jsx - Notification handling
- % % % % SearchProvider.js - Search functionality
- % % % Themes/ - Theme definitions
- % % % % ThemeContext.js - Light/dark mode theming
- % % % Animations/ - Animation components
- % % % util/ - Utility functions
- % % % % io_utils/ - API and data utilities
- % % % % data_structures/ - Custom data handling
- % % % App.js - Main application component
- % % % index.js - Application entry point

Data Flow

The application follows a unidirectional data flow pattern:

- 1. User interacts with a component, triggering an event handler
- 2. Event handler dispatches an action to the Redux store
- 3. For API operations:
- a. The appropriate API client (apiClient, flaskClient) sends the request
- b. Token interceptors automatically add authentication headers
- c. Response interceptors handle token expiration and errors
- 4. Redux reducers process actions and update state
- 5. Connected components re-render based on state changes
- 6. For real-time features, WebSockets provide immediate updates

Redux Store Structure

```
// Actual Redux store configuration from the codebase
export default configureStore({
  reducer: {
    searchResult: searchResult,
    userDetails: userDetails,
    search: Search,
    refreshMessages: refreshMessagesReducer,
    refreshSideBar: refreshSideBarReducer,
    refreshMailBox: refreshMailBoxReducer,
    auth: authReducer,
    scrollCursor: scrollCursorReducer
  },
})
```

Authentication Flow

The application uses JWT-based authentication with token persistence:

Login Process

- **1. User enters credentials in the Login component**
- **2. On form submission, credentials are sent to the authentication API**
- **3. Server validates credentials and returns a JWT token**
- **4. The Redux login action is dispatched with the token and user data**
- **5. Token is stored in localStorage and user state in Redux**
- **6. API client interceptors automatically use the token for subsequent requests**
- **7. User is redirected to the main application content**

Token Expiration Handling

- **1. API client interceptors detect 401 Unauthorized responses**
- **2. The logout action is dispatched to clear authentication state**
- **3. User is presented with the login screen**
- **4. localStorage and localForage data are cleared for security**

Authentication Code Implementation

```
// Authentication slice from actual codebase
const authSlice = createSlice({
  name: 'auth',
  initialState: {
    isAuthenticated: false,
    showLoginModal: false,
    user: null,
    token: null,
    isLoading: false,
  },
  reducers: {
    login: (state, action) => {
      state.isAuthenticated = true;
      state.user = action.payload.user;
      state.token = action.payload.token;

      // Store auth data in localStorage
      localStorage.setItem('isLoggedIn', 'true');
      localStorage.setItem('token', action.payload.token);
      if (action.payload.user?.userId) {
        localStorage.setItem('userId', action.payload.user.userId);
      }
    },
    logout: (state) => {
      state.isAuthenticated = false;
```

```
      state.user = null;
      state.token = null;

      // Clear stored auth data
      localStorage.removeItem('isLoggedIn');
      localStorage.removeItem('token');
      localStorage.removeItem('userId');
    }
    // ... other reducers
  }
});
```

Theme Management

The application implements a theme switching capability using React Context and Material UI's ThemeProvider:

- 1. ThemeContext.js defines the theme context and provider
- 2. Theme preferences (light/dark) are persisted in localStorage
- 3. The ThemeContextProvider wraps the application and provides theme values
- 4. Material UI theme is dynamically created based on the selected mode
- 5. Component styling is adjusted automatically based on the current theme

```
// Theme implementation from the actual codebase
export const ThemeContextProvider = ({ children }) => {
  const [mode, setMode] = useState('dark');

  // Load saved theme from localStorage
  useEffect(() => {
    const savedMode = localStorage.getItem('mode');
    if (savedMode === 'light' || savedMode === 'dark') {
      setMode(savedMode);
    }
  }, []);

  // Save theme changes to localStorage
  useEffect(() => {
    localStorage.setItem('mode', mode);
  }, [mode]);

  const toggleMode = () => {
    setMode((prev) => (prev === 'light' ? 'dark' : 'light'));
  };

  // Create Material UI theme based on mode
  const theme = useMemo(() => createTheme({
    palette: {
      mode,
      // Custom theme colors
      background: {
        default: mode === 'dark' ? '#000000' : '#ffffff',
        paper: mode === 'dark' ? 'rgba(0,0,0,0.6)' : '#ffffff',
      },
      // ... other theme settings
    }
  }), [mode]);

  return (
    <ThemeContext.Provider value={{ mode, toggleMode }}>
      <ThemeProvider theme={theme}>
        <CssBaseline enableColorScheme />
        {children}
      </ThemeProvider>
    </ThemeContext.Provider>
  );
};
```

Offline Support and Data Persistence

The application implements offline support and data persistence through:

- Service Workers: Registered for background processing and push notifications
- LocalForage: IndexedDB-based persistent storage for application data
- LocalStorage: For simple key-value storage of user preferences and tokens
- Push Notifications: Using the browser's Push API for notifications when the app is closed

```
// Service worker registration from the actual codebase
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/serviceworkers/
NotificationReceiver.js').then(
    registration => {
      navigator.serviceWorker.ready.then(readyRegistration => {
        if (Notification.permission === 'granted') {
          register(readyRegistration); // Register for push notifications
        } else if (Notification.permission === 'default') {
          Notification.requestPermission().then(permission => {
            if (permission === 'granted') {
              register(readyRegistration);
            }
          });
        }
      });
    }
  );
}
```

Routing and Navigation

The application uses React Router v6 for client-side routing with authentication protection:

- BrowserRouter provides the routing context for the application
- Authentication-based conditional rendering controls access to routes
- The Contents component contains all authenticated routes
- When not authenticated, the AccountIssue component is shown instead
- Error components handle specific error conditions like network issues

```
// Main routing structure from App.js
if (!isAuthenticated || showLoginModal) {
  return (
    <ThemeContextProvider>
      <AccountIssue
        loginState={isAuthenticated}
        setLoginState={(newState) => {
          // Login state management
        }}
      />
    </ThemeContextProvider>
  );
}

if (isAuthenticated) {
  return (
    <BrowserRouter>
      <ThemeContextProvider>
        <Contents
          setLogin={async (newState) => {
            // Login/logout handling
          }}
        />
      </ThemeContextProvider>
    </BrowserRouter>
  );
}
```

Error Handling

The application implements comprehensive error handling:

- Dedicated error components for specific error scenarios:
- • NetworkError: Displayed when internet connectivity is lost
- • EndpointNotAvailableError: Shown when API endpoints are unreachable
- • NotFoundError: For 404 errors when navigating to non-existent routes
- API error handling through axios interceptors
- Authentication error handling with automatic logout on token expiration
- Notifications for user feedback on operations

Real-time Communication

The application uses WebSockets for real-time features:

- WebSocket connection to wss://apis.vydeo.xyz/ws
- Real-time chat messaging between users
- Push notifications for new messages and content updates
- Message handlers for different types of real-time events
- Integration with Redux for state updates on WebSocket events

Security Considerations

The application implements the following security measures:

- JWT-based authentication with secure token storage
- Automatic token handling through API client interceptors
- Token expiration management and automatic logout
- Secure WebSocket communication with token authentication
- Environment-specific API endpoints for development and production
- Proper data clearing on logout to prevent information leakage