

Session Storage Implementation

Overview

The Dydra JSUI application uses a two-tier storage system for managing user sessions and authentication:

1. **Session Storage** (`Session` class): Persists the current logged-in account name using `localStorage`
2. **Auth Store** (`AuthStore` class): Maintains authentication tokens and configuration in memory only

Implementation Details

1. Session Class (`lib/models/session.js`)

The `Session` class manages the current user session using the browser's `localStorage` API.

Storage Key

```
const STORAGE_KEY = "dydra.session";
```

Stored State The session stores a minimal JSON object containing: - `accountName`: The friendly ID of the currently logged-in account (string or null)

Storage Format:

```
{  
  "accountName": "username"  
}
```

Implementation Constructor:

```
constructor() {  
  this.accountName = null;  
}
```

Initializes with no account name.

Load Method:

```
load() {  
  try {  
    const stored = JSON.parse(window.localStorage.getItem(STORAGE_KEY));  
    this.accountName = stored?.accountName || null;  
  } catch (error) {  
    this.accountName = null;  
  }  
}
```

```

    }
}

```

- Reads from `localStorage` using the key "dydra.session"
- Parses JSON and extracts `accountName`
- Handles errors gracefully (invalid JSON, missing key) by defaulting to `null`
- Called automatically when `AppState` is initialized

Save Method:

```

save() {
    window.localStorage.setItem(STORAGE_KEY, JSON.stringify({ accountName: this.accountName }));
}

```

- Serializes the session state to JSON
- Writes to `localStorage`
- Called automatically after login/logout operations

Login Method:

```

login(accountName) {
    this.accountName = accountName;
    this.save();
}

```

- Sets the account name
- Persists to `localStorage` immediately

Logout Method:

```

logout() {
    this.accountName = null;
    this.save();
}

```

- Clears the account name (sets to `null`)
- Persists the cleared state to `localStorage`

IsLoggedIn Method:

```

isLoggedIn() {
    return Boolean(this.accountName);
}

```

- Returns `true` if an account name is set, `false` otherwise

2. AuthStore Class (lib/auth_store.js)

The `AuthStore` class manages authentication tokens and configuration for multiple accounts **in memory only** (not persisted to storage).

Stored State (In-Memory)

```
{
  tokens: {
    "accountName1": {
      token: "Bearer ...",
      config: { ... },
      host: "https://dydra.com"
    },
    "accountName2": {
      token: "Bearer ...",
      config: { ... },
      host: "https://dydra.com"
    }
  }
}
```

Implementation Constructor:

```
constructor() {
  this.state = { tokens: {} };
}
```

Initializes with an empty tokens object.

Key Methods: - `setAuth(accountName, token, config, host)`: Stores authentication data for an account - `getToken(accountName)`: Retrieves the token for an account - `getConfig(accountName)`: Retrieves the configuration for an account - `getAuth(accountName)`: Retrieves the complete auth object (token, config, host) - `listAccounts()`: Returns array of all authenticated account names

Important: AuthStore data is **NOT persisted** to any storage mechanism. It exists only in memory and is lost on page reload.

3. Integration with AppState

The AppState class (`lib/app_state.js`) coordinates both storage systems:

```
export class AppState {
  constructor({ rdfClient } = {}) {
    this.session = new Session();
    this.session.load(); // Loads from localStorage on initialization
    this.authStore = new AuthStore(); // In-memory only
    // ... other initialization
  }
}
```

Initialization Flow: 1. `AppState` is created when the application starts (`app.js`) 2. `Session` is instantiated and immediately calls `load()` to restore the account name from `localStorage` 3. `AuthStore` is instantiated as an empty in-memory store 4. On page reload, the session account name is restored, but authentication tokens must be re-authenticated

State Lifecycle

Login Flow

1. User submits login form (`ui/pages/index.js` - `setupInlineLogin`)

```
const result = await authenticateAccount({ host, accountName, secret });
```
2. Authentication succeeds - API returns token and config
3. Store authentication data:

```
app.state.authStore.setAuth(accountName, result.token, result.config, result.baseUrl);
```
4. Update session:

```
app.state.session.login(accountName);
```

 - Sets `session.accountName`
 - Saves to `localStorage` via `save()`
5. Navigate to account page

Logout Flow

1. User clicks logout (`ui/app.js` - `handleLogout`)

```
handleLogout() {  
  this.state.session.logout();  
  this.router.navigate("/login", { replace: true });  
}
```
2. Session cleared:

```
logout() {  
  this.accountName = null;  
  this.save(); // Persists null to localStorage  
}
```
3. `AuthStore` remains in memory - tokens are not explicitly cleared, but become inaccessible since there's no session
4. Redirect to login page

Page Reload Flow

1. Application initializes (`app.js`)

```
const state = new AppState();
```

2. **Session loads from localStorage:**

```
this.session = new Session();  
this.session.load(); // Restores accountName from localStorage
```

3. **AuthStore is empty** - tokens are lost and must be re-authenticated

4. **Application checks session:**

- If `session.isLoggedIn()` returns `true`, user appears logged in
- However, `authStore` has no tokens, so API calls will fail
- User must re-authenticate to restore tokens

When State is Cleared

Session Storage (localStorage)

Cleared when: 1. **Explicit logout:** `session.logout()` sets `accountName` to null and saves 2. **Browser storage cleared:** User manually clears browser data or localStorage 3. **Private/Incognito mode:** localStorage is cleared when the session ends

NOT cleared when: - Page reload (persists across reloads) - Browser tab closed (persists across tabs in same origin) - Application errors (persists unless explicitly cleared)

AuthStore (In-Memory)

Cleared when: 1. **Page reload:** All in-memory state is lost 2. **Browser tab closed:** JavaScript context is destroyed 3. **Application restart:** New `AppState` instance creates new empty `AuthStore`

NOT explicitly cleared: - Logout does not clear `AuthStore` (becomes inaccessible but remains in memory until page reload) - No explicit cleanup method exists

Storage Characteristics

localStorage (Session)

- **Persistence:** Survives page reloads, browser restarts (until explicitly cleared)
- **Scope:** Shared across all tabs/windows of the same origin
- **Capacity:** ~5-10MB (browser dependent)
- **Lifetime:** Until explicitly cleared or browser data is cleared
- **Security:** Accessible to JavaScript in same origin, vulnerable to XSS

In-Memory (AuthStore)

- **Persistence:** Lost on page reload
- **Scope:** Per-tab/window instance
- **Capacity:** Limited by available RAM
- **Lifetime:** Until page reload or tab close
- **Security:** More secure (not persisted), but tokens still accessible to JavaScript

Design Rationale

Why Session Uses localStorage

- **User Experience:** Users remain “logged in” after page reload
- **Convenience:** Avoids requiring re-authentication on every page load
- **State Persistence:** Maintains which account was last used

Why AuthStore is In-Memory Only

- **Security:** Tokens are not persisted to disk, reducing exposure
- **Fresh Authentication:** Forces re-authentication on page reload, ensuring tokens are current
- **Multi-Account Support:** Allows multiple accounts to be authenticated simultaneously during a session

Trade-offs

Current Design: - User appears logged in after reload (good UX) - Tokens not persisted (better security) - User must re-authenticate after reload (may be unexpected) - Session state and auth state can become out of sync

Potential Issues: 1. **State Mismatch:** `session.accountName` may indicate logged-in status, but `authStore` has no tokens 2. **Silent Failures:** API calls may fail silently if tokens are missing 3. **User Confusion:** User may see “logged in” UI but operations fail

Usage Examples

Checking Login Status

```
if (app.state.session.isLoggedIn()) {  
  // User appears logged in  
  const accountName = app.state.session.accountName;  
}
```

Getting Authentication Token

```
const accountName = app.state.session.accountName;  
const token = app.state.getAuthToken(accountName);
```

```
if (!token) {  
  // User needs to re-authenticate  
}
```

Getting Full Auth Context

```
const auth = app.state.getAuthContext(accountName);  
if (auth) {  
  const { token, config, host } = auth;  
  // Use for API calls  
}
```

Logout

```
app.state.session.logout();  
// Session cleared, but authStore tokens remain in memory until reload
```

Recommendations for Improvement

1. **Add AuthStore Persistence:** Consider persisting tokens to `sessionStorage` (cleared on tab close) for better UX
2. **Synchronize State:** Clear `authStore` on logout to prevent state mismatch
3. **Token Validation:** Check token validity on page load and clear session if invalid
4. **Explicit Cleanup:** Add `authStore.clear()` method and call it on logout
5. **Error Handling:** Better handling when session exists but tokens are missing