Implementation Documentation

Software Architecture

Overview

The Dydra JSUI application is a single-page application (SPA) built with vanilla JavaScript using ES6 modules. The architecture follows a component-based, layered design pattern with clear separation of concerns:

1. **Presentation Layer**: UI components and page renderers
2. **Application Layer**: Application state management and routing
3. **Domain Layer**: Business logic and model classes
4. **Persistence Layer**: Data access and synchronization

Architecture Diagram

```
                      Browser Environment



                 Presentation Layer (UI)

         Pages           Components       Layout
        (Renderers)      (Reusable)       (Shell)




             Application Layer (State & Routing)

          Router         AppState           App
        (Navigation)     (State Mgmt)    (Orchestr)




               Domain Layer (Business Logic)

         Models         Replication         Auth
       (Account,          Manager          Store
        Repo, etc.)
```

```
           Persistence Layer (Data Access)

        Adapter        RDF Store
       (Interface)       Adapter




                      Dydra API
                      (External)
```

Component Dependencies

Core Application Components

1. Application Entry Point (`app.js`)   **Location**: `/app.js`

Dependencies: - `Router` from `./router.js` - `AppState` from `./lib/app_state.js` - `APP_CONFIG` from `./lib/config.js` - `App` from `./ui/app.js` - `buildRoutes` from `./ui/routes.js`

Responsibilities: - Initializes application state - Creates application instance - Sets up routing - Starts the router

Dependency Flow:

```
app.js
   > Router (handles navigation)
   > AppState (manages global state)
   > App (orchestrates UI rendering)
   > buildRoutes (defines route handlers)
```

2. Router (`router.js`)   **Location**: `/router.js`

Dependencies: None (standalone module)

Responsibilities: - Client-side routing with History API or hash-based fallback - Route matching with parameter extraction - Browser history management - Link click interception

Key Features: - Supports parameterized routes (e.g., `/account/:account_name`) - Maintains internal history stack - Handles browser back/forward navigation - Supports base path configuration

3. Application State (`lib/app_state.js`)   **Location**: `/lib/app_state.js`

Dependencies: - `Account` from `./models/account.js` - `Repository` from `./models/repository.js` - `Query` from `./models/query.js` - `Invitation` from `./models/invitation.js` - `Session` from `./models/session.js` - `AuthStore` from `./auth_store.js` - `ReplicationManager` from `./replication/replication_manager.js` - `RdfStoreAdapter` from `./persistence/rdf_store_adapter.js` - `sampleData` from `./sample_data.js`

Responsibilities: - Manages global application state - Coordinates data access through adapters - Tracks open accounts and repositories - Manages replication trackers for account and repository configurations - Provides caching layer with fallback to sample data

Dependency Flow:

```
AppState
   > Session (user session management)
   > AuthStore (authentication tokens)
   > RdfStoreAdapter (data persistence)
   > ReplicationManager (state synchronization)
   > Models (Account, Repository, Query, Invitation)
```

4. Application UI (`ui/app.js`)   **Location**: `/ui/app.js`

Dependencies: - `LayoutView`, `HomeLayoutView` from `./components/layout.js` - `NotFoundPage` from `./pages/index.js` - `APP_CONFIG` from `../lib/config.js`

Responsibilities: - Orchestrates page rendering - Manages layout selection (standard vs. home layout) - Handles tab initialization and management - Manages location bar updates - Coordinates pane navigation (back/forward buttons)

Dependency Flow:

```
App
   > LayoutView / HomeLayoutView (page structure)
   > Pages (content renderers)
   > Router (for navigation)
```

UI Components

5. Pages (`ui/pages/`)   **Location**: `/ui/pages/`

Base Class: `BasePage` (`base_page.js`)

Dependencies: - `AppState` (via constructor) - Various UI components as needed

Page Types: - `HomePage`: Landing page - `LoginPage`, `SignupPage`: Authentication pages - `AccountShowPage`, `AccountEditPage`, `AccountNewPage`: Account management - `RepositoryShowPage`, `RepositoryEditPage`, `RepositoryNewPage`: Repository management - `SparqlPage`: SPARQL query interface - `QueryShowPage`, `QueryEditPage`, `QueryNewPage`: View/query management - `InvitationsNewPage`, `InvitationsIndexPage`: Invitation management - `NotFoundPage`: 404 error page

Responsibilities: - Render page-specific content - Provide sidebar content - Handle page-specific interactions - Define page metadata (title, body class)

6. Layout Components (`ui/components/`) **Location**: `/ui/components/`

Components: - `layout.js`: Main layout with header, content, sidebar, footer - `header.js`: Navigation header - `footer.js`: Page footer - `navigation.js`: Navigation menu - `flashes.js`: Flash message notifications

Dependencies: - Application state for authentication status - Router for navigation links

Domain Models

7. Model Classes (`lib/models/`) **Location**: `/lib/models/`

Models: - `account.js`: Account entity - `repository.js`: Repository entity - `query.js`: Query/View entity - `invitation.js`: Invitation entity - `session.js`: User session entity - `import_job.js`: Import job entity

Responsibilities: - Represent domain entities - Provide business logic methods - Handle data transformation

Dependencies: None (pure domain objects)

8. Authentication (`lib/auth.js`, `lib/auth_store.js`) **Location**: `/lib/auth.js`, `/lib/auth_store.js`

Dependencies: - Browser Fetch API - Browser Storage API (localStorage/sessionStorage)

Responsibilities: - Authenticate users against Dydra API - Store and retrieve authentication tokens - Build authentication headers (Basic/Bearer) - Normalize host URLs

Key Functions: - `authenticateAccount()`: Performs authentication and returns token - `AuthStore.getToken()`: Retrieves stored token - `AuthStore.setToken()`: Stores token

Replication System

9. Replication Manager (`lib/replication/replication_manager.js`) **Location**: `/lib/replication/replication_manager.js`

Dependencies: - `GraphObject` from `./graph-object.js`

Responsibilities: - Manages object replication lifecycle - Creates and manages GraphObject instances - Handles state synchronization - Replaces object state when needed

Key Concepts: - Uses `GraphObject` as base class for replicated entities - Tracks persistent and editable properties - Manages object state transitions

10. Graph Object (`lib/replication/graph-object.js`) Location: `/lib/replication/graph-object.js`

Dependencies: None (uses JavaScript Proxy API)

Responsibilities: - Implements JDO/JPA-like state machine for object lifecycle - Tracks property changes via delta maps - Generates patches for remote synchronization - Supports rollback and rollforward operations

State Machine: - `new`: Unattached object - `clean`: Attached, unmodified - `modified` (dirty): Attached, has changes - `deleted`: Marked for deletion

Key Features: - Uses JavaScript Proxy to intercept property access - Maintains delta maps: `{property: [newValue, oldValue]}` - Generates patches: `{post: [], put: [], delete: []}`

Persistence Layer

11. Persistence Adapter (`lib/persistence/adapter.js`) Location: `/lib/persistence/adapter.js`

Dependencies: None (abstract interface)

Responsibilities: - Defines interface for data persistence operations - Provides abstract methods for CRUD operations

Interface Methods: - `getAccountByName(name)` - `listAccounts()` - `listRepositories(accountName)` - `getRepository(accountName, repositoryName)` - `listQueries(accountName, repositoryName)` - `getQuery(accountName, repositoryName, queryName)` - `listInvitations()`

12. RDF Store Adapter (`lib/persistence/rdf_store_adapter.js`) Location: `/lib/persistence/rdf_store_adapter.js`

Dependencies: - `PersistenceAdapter` from `./adapter.js` - External RDF client (injected via constructor)

Responsibilities: - Implements PersistenceAdapter interface - Delegates to RDF client for data operations - Handles cases where RDF client is unavailable

Dependency Injection: - Accepts `rdfClient` in constructor - Falls back gracefully when client is not provided

SPARQL Editor Integration

13. SPARQL Editor (`js/sparql-editor.js`) Location: `/js/sparql-editor.js`

Dependencies: - YASQE (Yet Another SPARQL Query Editor) - external library - CodeMirror (via YASQE) - external library - Browser Fetch API

Responsibilities: - Provides unified SPARQL query editor component - Handles query execution against SPARQL endpoints - Manages multiple result formats (JSON, XML, SVG, HTML, CSV, TSV, RDF formats) - Supports parameterized queries - Maintains query execution history - Provides event logging

Key Features: - Standalone mode (with access token) - Integrated mode (with DydraClient) - Collapsible interface - Tabbed result display - Media type selection - Query save/load functionality

Dependency Relations

Dependency Graph

```
app.js

   > router.js (no dependencies)

   > lib/app_state.js

         > lib/models/*.js (no dependencies)
         > lib/auth_store.js (no dependencies)
         > lib/replication/replication_manager.js
               > lib/replication/graph-object.js (no dependencies)
         > lib/persistence/rdf_store_adapter.js
                > lib/persistence/adapter.js (no dependencies)

   > lib/config.js (no dependencies)

   > ui/app.js

         > ui/components/layout.js
               > ui/components/header.js
               > ui/components/footer.js
               > ui/components/navigation.js

         > ui/routes.js
               > ui/pages/*.js
                     > ui/pages/base_page.js
```

Circular Dependencies

None identified. The architecture is designed to avoid circular dependencies through: - Unidirectional data flow (top-down) - Dependency injection for adapters - Event-based communication where needed

External Dependencies

Browser APIs

- **Fetch API**: For HTTP requests
- **History API**: For client-side routing
- **Proxy API**: For GraphObject property interception
- **Storage API**: For session and token storage
- **DOM API**: For UI manipulation

External Libraries (Loaded via Script Tags)

1. **Bootstrap CSS** (`css/bootstrap*.css`)
   - **Purpose**: UI styling and layout
   - **Usage**: Global CSS classes for buttons, forms, tables, etc.
   - **Dependency Level**: Presentation layer
2. **jQuery UI** (`stylesheets/jquery-ui/`)
   - **Purpose**: UI widgets and interactions
   - **Usage**: Dialog boxes, date pickers, etc.
   - **Dependency Level**: Presentation layer
3. **YASQE** (Yet Another SPARQL Query Editor)
   - **Purpose**: SPARQL query editing with syntax highlighting
   - **Usage**: Integrated in `js/sparql-editor.js`
   - **Dependency Level**: Feature-specific (SPARQL editor)
4. **CodeMirror** (via YASQE)
   - **Purpose**: Code editing engine
   - **Usage**: Used by YASQE for editor functionality
   - **Dependency Level**: Indirect (via YASQE)

Significant Implementation Techniques

1. Client-Side Routing

Technique: Custom router implementation with History API fallback

Implementation: - Uses `window.history.pushState()` for modern browsers - Falls back to hash-based routing (`window.location.hash`) for file:// protocol - Compiles route patterns to regular expressions for matching - Extracts route parameters from URL segments

Key Code (`router.js`):

```
compileRoute(path) {
  const keys = [];
  const pattern = path
    .split("/")
    .map((segment) => {
      if (segment.startsWith(":")) {
        keys.push(segment.slice(1));
        return "([^/]+)";
      }
      return segment.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
    })
    .join("/");
  return { regex: new RegExp(`^${pattern}$`), keys };
}
```

2. State Management with Replication

Technique: JDO/JPA-like state machine with Proxy-based change tracking

Implementation: - Uses JavaScript Proxy to intercept property access - Maintains state: `new`, `clean`, `modified`, `deleted` - Tracks changes in delta maps: `{property: [newValue, oldValue]}` - Generates patches for remote synchronization

Key Code (`lib/replication/graph-object.js`):

```
set(target, name, value) {
  if (handler._replicator && properties.includes(name)) {
    const oldValue = target[name];
    if (oldValue != value) {
      const deltas = handler._deltas;
      if (!deltas) handler._deltas = {};
      deltas[name] = [value, oldValue];
      handler._state = GraphObject.stateModified;
    }
  }
  target[name] = value;
  return true;
}
```

3. Component-Based UI Rendering

Technique: Class-based page and component system

Implementation: - Base `BasePage` class with lifecycle methods - Pages implement `renderContent()`, `renderSidebar()`, `getTitle()` - Layout components compose pages into final HTML - Async rendering support for data fetching

Key Code (`ui/pages/base_page.js`):

```
export class BasePage {
  async renderContent() { return ""; }
  async renderSidebar() { return ""; }
  getTitle() { return "Dydra"; }
  async afterRender() { return; }
}
```

4. Dependency Injection for Adapters

Technique: Constructor injection for external dependencies

Implementation: - `RdfStoreAdapter` accepts `rdfClient` in constructor - `AppState` accepts optional `rdfClient` for adapter initialization - Graceful fallback when dependencies are unavailable

Key Code (`lib/persistence/rdf_store_adapter.js`):

```
constructor({ rdfClient }) {
  super();
  this.rdfClient = rdfClient;
}

async getAccountByName(name) {
  if (!this.rdfClient) return null;
  return this.rdfClient.getAccount(name);
}
```

5. Tab Management System

Technique: Stateful tab tracking with DOM synchronization

Implementation: - Tracks open accounts and repositories in `AppState` - Maintains tab state separate from route state - Synchronizes tab UI with application state - Supports closing tabs and navigation updates

Key Code (`ui/app.js`):

```
addOpenAccount(accountName) {
  if (!accountName) return;
  this.openAccounts.add(accountName);
}

closePane({ tabType, tabId, accountName, repositoryName }) {
  if (tabType === "account" && accountName) {
    this.state.removeOpenAccount(accountName);
  }
  // ... handle navigation
}
```

6. SPARQL Editor Integration

Technique: Unified editor component with multiple integration modes

Implementation: - Standalone mode: Direct API calls with access token - Integrated mode: Uses DydraClient for API access - Supports multiple result formats via media type selection - Tabbed result display with execution history

Key Features: - YASQE integration for syntax highlighting - Parameter substitution for parameterized queries - Event logging for debugging - Collapsible interface for space efficiency

Functional Dependencies on External Libraries

1. Bootstrap CSS Framework

Purpose: UI styling and responsive layout

Dependencies: - Bootstrap CSS files in `/css/` - Bootstrap theme files - Glyphicons font files in `/fonts/`

Usage: - Global CSS classes applied throughout the application - Grid system for layout - Form styling - Button and navigation components

Impact: High - Core styling depends on Bootstrap classes

2. jQuery UI

Purpose: UI widgets and interactions

Dependencies: - jQuery UI CSS in `/stylesheets/jquery-ui/` - jQuery UI theme (Aristo) with images

Usage: - Dialog components - Date picker widgets - UI theme styling

Impact: Medium - Used for specific UI components

3. YASQE (SPARQL Editor)

Purpose: SPARQL query editing

Dependencies: - YASQE library (loaded via script tag) - CodeMirror (bundled with YASQE)

Usage: - Integrated in `js/sparql-editor.js` - Provides syntax highlighting - Autocomplete functionality - Query validation

Impact: High - Critical for SPARQL query functionality

Integration Points:

```
// In sparql-editor.js
queryEditor = createQueryEditor(queryEditorContainer, {
  useYasqe: options.useYasqe !== false,
  initialQuery: initialQuery,
  enableAutocomplete: options.enableAutocomplete !== false,
  prefixes: schemaData.prefixes || {},
  endpoint: viewUrl
});
```

4. Browser APIs

Fetch API

- **Purpose**: HTTP requests to Dydra API
- **Usage**: All API communication
- **Impact**: Critical - No fallback available

History API

- **Purpose**: Client-side routing
- **Usage**: Navigation and URL management
- **Fallback**: Hash-based routing for file:// protocol

Proxy API

- **Purpose**: Property interception for GraphObject
- **Usage**: Change tracking in replication system
- **Impact**: High - Core to replication functionality

Storage API

- **Purpose**: Session and token persistence
- **Usage**: localStorage/sessionStorage for auth tokens
- **Impact**: Medium - Affects session persistence

Data Flow

Authentication Flow

```
User Input (credentials)


authenticateAccount() (lib/auth.js)

   > Normalize host URL
   > Build auth header (Basic/Bearer)
   > Fetch /system/accounts/<account>/configuration

         > Success: Store token in AuthStore

               > Update Session

                     > Redirect to account page

         > Failure: Display error message
```

Page Rendering Flow

```
Route Change


Router.findRoute()

   > Extract route parameters
   > Call route handler


App.renderPage(Page)

   > Page.renderContent() [async]
   > Page.renderSidebar() [async]
   > Page.getTitle()
   > Layout.render({ content, sidebar, title })

         > Update DOM

               > Page.afterRender() [async]
```

Data Synchronization Flow

```
User Edit (e.g., account field)


GraphObject property setter (via Proxy)

   > Check if property is persistent
   > Record delta: {property: [newValue, oldValue]}
   > Update state to 'modified'


User Clicks Save

   > GraphObject.asPatch()
        > Generate patch: {post: [], put: [], delete: []}

   > POST to API endpoint

         > Success: setStateClean()
         > Failure: rollback() deltas
```

Configuration

Application Configuration (`lib/config.js`)

Configuration Options: - `baseHost`: Main application host (default: “dydra.com”) - `blogHost`: Blog host (default: “blog.dydra.com”) - `docsHost`: Documentation host (default: “docs.dydra.com”) - `requireSignupInvite`: Whether signup requires invitation (default: true) - `showAccountBalances`: Whether to display account balances (default: false) - `basePath`: Base path for deployment in subdirectories (default: “/javascripts/jsui”)

Usage: Imported as `APP_CONFIG` throughout the application

Error Handling Strategy

Authentication Errors

- Display user-friendly error messages
- Log detailed errors to console for debugging
- Preserve user input for retry

API Errors

- Parse error responses (JSON or text)
- Display contextual error messages
- Provide retry mechanisms where appropriate

Routing Errors

- Display 404 page for unknown routes
- Maintain navigation context

Replication Errors

- Rollback changes on failure
- Restore previous state
- Notify user of synchronization issues

Testing Considerations

Unit Testing

- Test route matching and parameter extraction
- Test state machine transitions in GraphObject
- Test delta map generation and patch creation

Integration Testing

- Test authentication flow

- Test page rendering with mock data
- Test API adapter with mock RDF client

End-to-End Testing

- Test complete user workflows
- Test navigation and routing
- Test SPARQL query execution

Performance Optimizations

1. **Lazy Loading**: Pages and components loaded on demand
2. **Caching**: Sample data cache for offline/fallback scenarios
3. **Event Log Limiting**: Maximum 100 entries to prevent memory issues
4. **DOM Updates**: Batch DOM updates where possible
5. **Memory Management**: Cleanup of event listeners and DOM references

Security Considerations

1. **Token Storage**: Tokens stored in memory or secure storage
2. **Input Validation**: User input validated before API calls
3. **HTTPS**: All API communications over HTTPS
4. **XSS Prevention**: HTML escaping in user-generated content
5. **CSRF Protection**: Relies on API-level CSRF protection

Deployment Considerations

1. **Base Path Configuration**: Supports deployment in subdirectories
2. **Asset Paths**: Relative paths for CSS, images, fonts
3. **Hash Routing Fallback**: Works with file:// protocol for local development
4. **Browser Compatibility**: Requires ES6+ JavaScript support