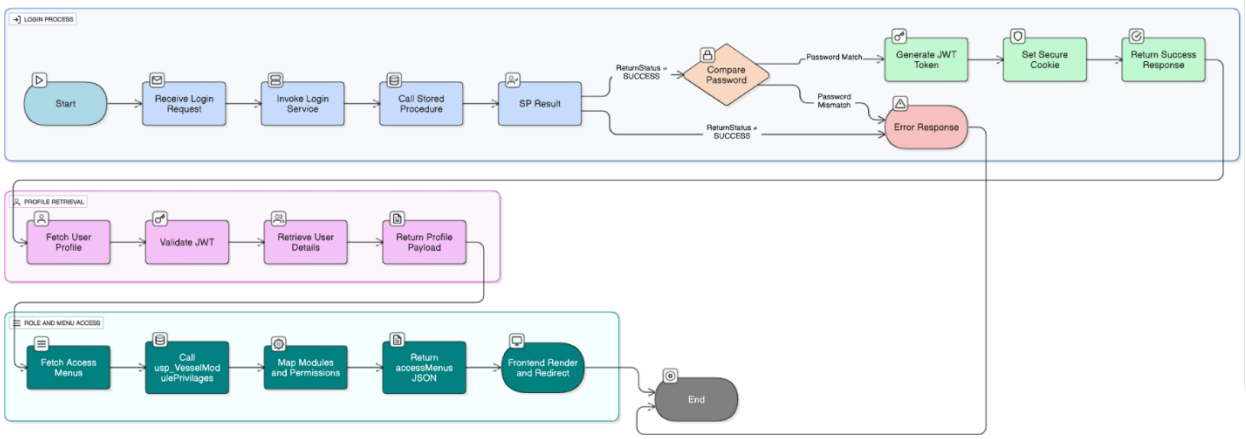
SeaMate Lite

Authentication & Authorization Backend Flow

Table of Contents



1. Login Process Overview

1.1 Password Hashing

- Passwords are hashed using **scrypt** in C# for consistency.
- Comparisons use `ScryptEncoder.Compare(...)`; no SQL-side hashing.

1.2 API Trigger

- **Endpoint:** `POST /api/auth/login`
- **Payload:**

```
{
  "empNo": "010255",
  "password": "user@123"
}
```

1.3 Backend Validation Flow

- Controller → `ILoginService` → `ILoginRepository`.
- Repository calls `[CM].[usp_login_lite]` with:
  - `@EMPNO`
  - `@SETTING_VALUE`
  - `@ReturnStatus` (output)
- SP performs:
  - User existence & `ONBOARD` status check
  - Account lock/unlock (`ACCOUNT_LOCK_DAYS`)
  - Returns user details (excluding password)
- C# compares password via `ScryptEncoder.Compare(...)`.

1.4 Password Comparison Decision

- After SP return, ReturnStatus = 'SUCCESS':
  - Compare input password with stored hash using ScryptEncoder.Compare(...)
- If **password matches**, proceed to generate JWT.
- If **password does not match**, return error with status code 401 (unauthorized).

1.5 JWT Token Generation

- On `ReturnStatus = 'SUCCESS'`:
  - Generate JWT (claims: `userId`, `role`, `menus`)
  - Store in Secure, Http Only, SameSite cookie.

1.6 Cookies Handling

- JWT stored in:
  - **Secure**
  - **HttpOnly**
  - **SameSite** cookie
- Prevents XSS and CSRF

1.7 API Result Handling

SP Code	HTTP Status
`SUCCESS`	200
`USER_NOT_FOUND_OR_INACTIVE`	401
`ACCOUNT_LOCKED`	423
`ERROR_*`	500

1.8 API Post-Login Handling

- After successful login and JWT issuance:
  - Client stores the JWT cookie automatically.
  - Frontend requests initial user context via `GET /api/user/profile`.
  - Server validates token, retrieves user details (name, empNo, roles).
  - Responds with profile payload to initialize UI (e.g., display username).

1.9 2G Environment Optimizations

- Minimal payload: `empNo`, `password` only
- SP returns essential data only
- Password compare in C# to keep SQL lightweight
- Single JWT issuance; no file transfers

1.10 Action Item

Action Item: Confirm source table for `companyPhoto` field.

2. User Role Access

2.1 Overview

This process determines accessible modules based on `EMP_ID` and position:

- **Masters**: Full access
- **Officers & Crew**: Limited access

Role–menu mapping is database-driven (no hardcoding).

2.2 Backend Role Determination Flow

1. **Input Mapping:** `USER_ID`, `APPLICATION_ID`
2. **SP Call:**

```
EXEC [CM].[usp_VesselModulePrivilages]
  @UserId = @USER_ID,
  @ApplicationId = @APPLICATION_ID;
```

3. **SP Logic:**
    - Map `EMP_ID` → `POSITION_ID` → `POSITION_GROUP`
    - Retrieve `PROFILE_ID` (Master/Officer/Crew)
    - Join role–module tables for allowed modules & privileges
    - Handle dynamic menu renaming (e.g. “Monthly Allotment” → “Voluntary Allotment”)

2.3 Menu Access

SP returns:

- `MODULE_ID`, `MODULE_NAME`
- Permission flags: `VIEW_ALLOWED`, `ADD_ALLOWED`, etc.
- `MENU_ORDER`, `MENU_PARENT_ID`

Mapped to `accessMenus` and `defaultPage`.

2.4 API Result Handling

```
{
  "accessMenus": [
    { "moduleId": 10, "moduleName": "Dashboard", "viewAllowed":
true, "addAllowed": false, "children": [] }
  ],
  "defaultPage": "/crew-list"
}
```

HTTP statuses follow Section 1.

2.5 API Layer Processing

- **Endpoint:** `GET /api/auth/access`
- Controller → Service → Repository → SP → JSON mapping

2.6 Frontend Behaviour

- Dynamically render navigation from `accessMenus`
- Redirect to `defaultPage` after login

2.7 Notes

- Currently reuses Office module SP; consider Lite SP for optimization.
- All mappings remain backend-driven.