TECHNICAL DOCUMENTATION: [PRIYANSH KHUNT]

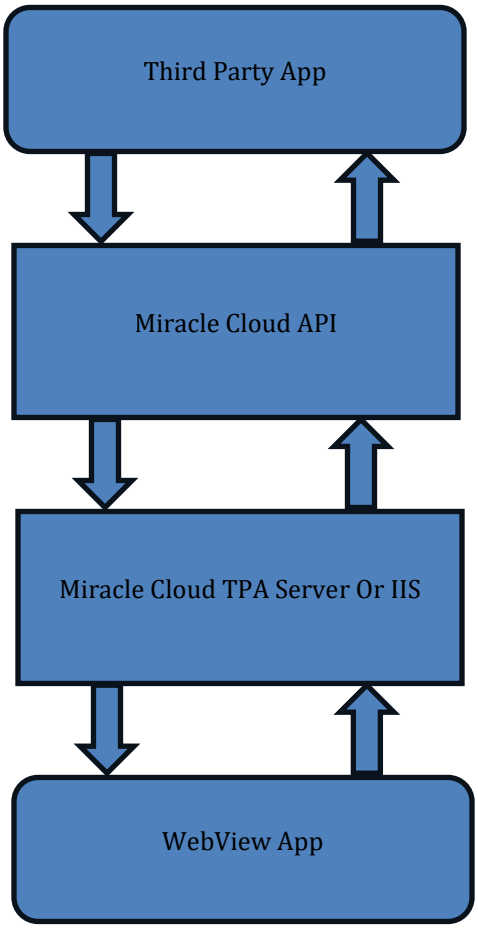
1. GOALS

The primary objectives of this system are:

- **Secure Token Transmission:**
  Securely transmit authentication tokens and user-specific data (DTO) from the API to the GUI.
- **Login Flow Confirmation:**
  Ensure the login API waits for GUI confirmation before finalizing the login process.

THIRD PARTY API PROCESSING USING WEBVIEW

- Process Flow Diagram



Here we assume that **MCGUI** is being running on WebView App

Following is the flow that what we want to archive

```
API1 (login)
    username, password
    using swagger
        token, role --> GUI (web application)
        return response
API2(getDTO)
    return to GUI with updated DTO

GUI
    setCookie/setItem
    call API2
    response from API2
    back to caller
```

2. APPROACH : MANUAL **URL REDIRECTION**

2.1 WORKFLOW

1. **API Generates Token:**

   After a successful login, the API generates a JWT token.

2. **Redirection URL:**

   The API returns a URL containing the token as a query parameter:

   https://gui.com?token=eyJhbGci...

3. **Manual Step:**

   The user **manually copies the URL and opens it** in a browser.

4. **GUI Processes Token:**

   The **GUI extracts the token** from the URL and stores it in **cookies** or **sessionStorage**.

2.2 PROS AND CONS

Pros:

- Simple to implement.

- No dependency on real-time communication protocols.

Cons:

- **Security Risk:** Tokens are exposed in URLs (visible in browser history, logs).
- **Poor User Experience:** Requires manual steps, which can disrupt workflow.
- **Limited Scalability:** Not suitable for automated or large-scale systems.

Why Not Used:

Manual intervention is error-prone and insecure, making this approach unsuitable for production environments.

---

3. APPROACH : **HTTP CLIENT**

3.1 WORKFLOW

1. **API Generates Token:**
   A JWT token is created upon successful login.
2. **Redirection URL:**
   The API returns a redirection URL containing the token as a query parameter.
3. **HTTP Client Call:**
   The GUI automatically follows the redirection using **HttpClient**:

```
var response = await httpClient.GetAsync(redirectUrl);
```

4. **GUI Processes Token:**
   The GUI extracts the token from the response and stores it in **cookies** or **sessionStorage**.

3.2 PROS AND CONS

Pros:

- Automates token handling.
- No manual user involvement.

Cons:

- **Limited Persistence:** Cookies are stored server-side, not in the browser.

- **Token Loss:** Tokens are lost when the GUI is refreshed.

- **Error Handling Issues:** Always returns HTTP 200, making errors harder to detect.

- **Cross-Domain Issues:** Requires complex CORS configurations.

3.3 WHY NOT USED

- Token getting stored in server context so if user refresh his tab he don't get cookie.

4. APPROACH : **WEBSOCKET**

- **WebSocket Connection:** GUI establishes a persistent connection to **ws://localhost:8181**.
- **Token Transmission:** API sends the token via WebSocket after successful login.

- **GUI Processing:** Stores token → Fetches DTO → Confirms via WebSocket (**DTO_CONFIRMED:token:json**).

Why Chosen:

- Best balance of security

- Automation

- Cross-domain support.

5. CURRENT IMPLEMENTATION: WEBSOCKET WORKFLOW

API Sequence:

1. User logs in → API generates JWT token → Sends token via WebSocket. [Login API waiting]

GUI Sequence:

1. Receives token → Stores it → Fetches DTO → Send Updated DTO via WebSocket. → DTO shared as Response of Login API

Timeout Handling:

- API waits 30 seconds for GUI confirmation of DTO.

5.1 PROS AND CONS

Pros:

- **Real-Time Updates:** Immediate token transmission without polling.

- **Improved Security:** Tokens aren't exposed in URLs.

- **Cross-Domain Support:** Works across domains with proper CORS configuration.

- **Scalable:** Efficient handling of multiple clients.

Cons:

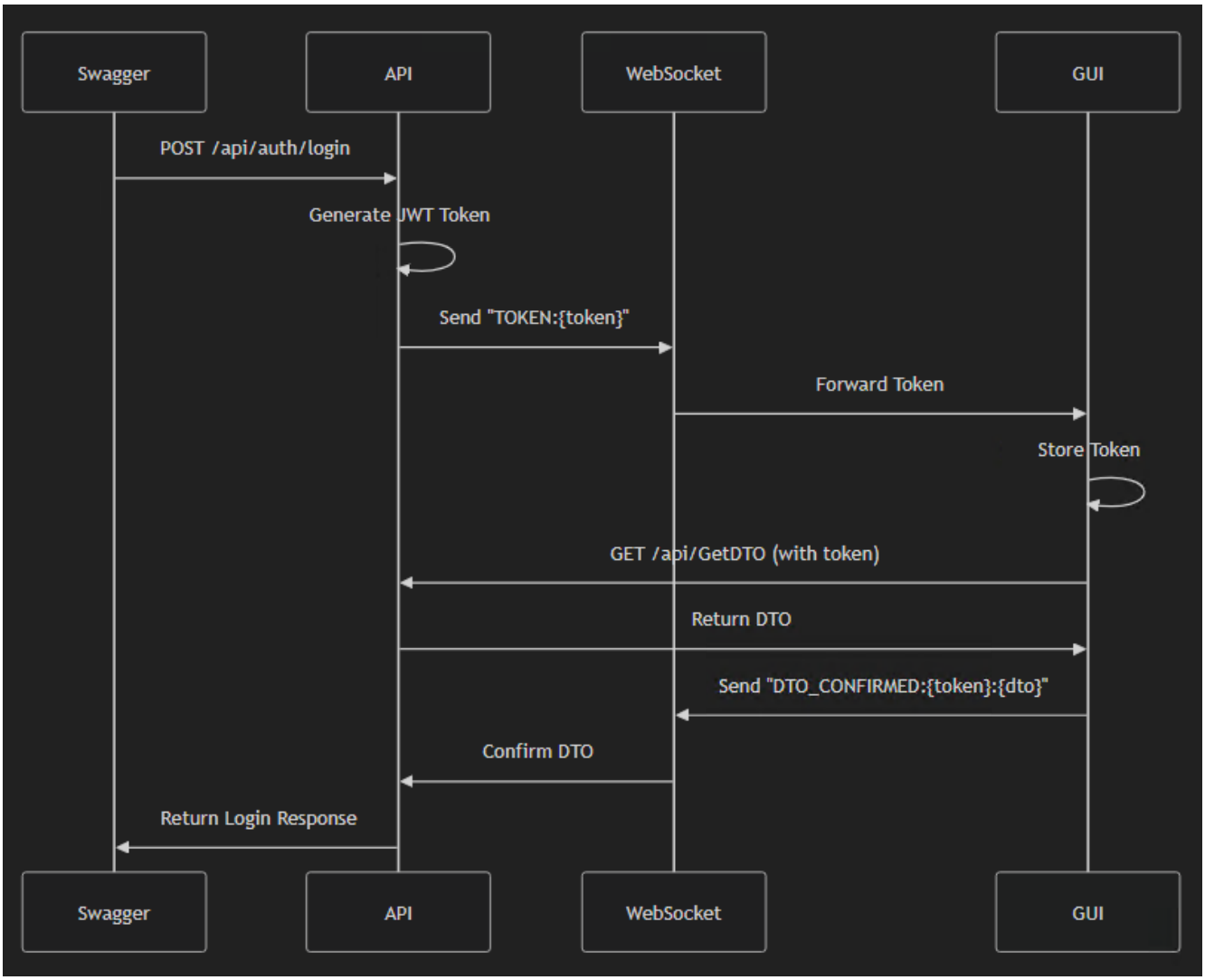
- **Complex Setup:** Requires WebSocket server management.

- **Firewall Restrictions:** WebSocket ports (e.g., 8181) might be blocked.

- **Resource Usage:** Persistent connections consume more server resources.

4.2 WHY IT'S THE BEST CHOICE FOR CROSS-DOMAIN

- **Cross-Domain Flexibility:** WebSocket connections bypass traditional CORS limitations.
- **Real-Time Communication:** Ideal for dynamic applications needing live updates.
- **Enhanced Security:** Supports encrypted communication via WSS (WebSocket Secure).



5. CROSS-DOMAIN CONSIDERATIONS

CORS Configuration:

```
builder.Services.AddCors(options =>
  options.AddPolicy("AllowAll", policy =>
    policy.AllowAnyOrigin()
      .AllowAnyMethod()
      .AllowAnyHeader()));
```

WebSocket Security:

- Use wss:// in production.

- Validate origins on the WebSocket server.

6. PROS AND CONS ANALYSIS

CRITERIA	MANUAL URL REDIRECTION	HTTP CLIENT	WEBSOCKET
SECURITY	❌ (TOKEN IN URL)	✅	✅ (ENCRYPTED)
AUTOMATION	❌ (MANUAL STEPS)	✅	✅
CROSS-DOMAIN	✅	❌ (CORS)	✅
SCALABILITY	❌	⚠️ (LIMITED)	✅ (HIGH)
USER EXPERIENCE	❌	⚠️ (COOKIE ISSUES)	✅

7. FUTURE RECOMMENDATIONS FOR PRODUCTION

WebSocket Enhancements:

- Use Azure SignalR or Redis for scaling WebSocket connections.

Security:

- Use HTTPS/WSS exclusively.

Load Balancing:

- Use sticky sessions for WebSocket connections.

Monitoring:

- Track message rates and connection lifetimes.

Fallback Mechanism:

- Use long-polling for clients blocking WebSocket.

Token Expiry:

- Implement refresh tokens for long-lived sessions.

8. IMPLEMENTING WEBSOCKET IN ASP.NET 4.7

8.1 CHALLENGES

- **No Native Support:**
  ASP.NET 4.7 lacks built-in WebSocket support compared to ASP.NET Core.
- **Complex Configuration:**
  Requires manual setup for handling WebSocket connections.
- **Performance Constraints:**
  Less efficient than the WebSocket features available in ASP.NET Core.

8.2 POSSIBLE SOLUTIONS

- **Use SignalR for ASP.NET 4.7:**
  SignalR offers WebSocket-like functionality with built-in fallback mechanisms.
- **Third-Party Libraries:**
  Libraries like **Fleck** or **WebSocketSharp** can provide WebSocket functionality.
- **Upgrade to .NET Core:**
  Migrating to ASP.NET Core offers better WebSocket support and performance.