## Use Case 1: Multiple Users Logging In Simultaneously

### Problem:

* Multiple users log in via Swagger, and tokens are sent to the same GUI.
* Cookies get overwritten, causing conflicts between users.

### Solution:

#### Use Session-Specific Cookies:

Store the token in a session-specific cookie (e.g., socketId\_token).

// Store token with session-specific cookie  
document.cookie = `socket\_${socket.id}\_token=${token}; path=/;`;

#### Socket ID as Cookie Identifier:

Associate the token with the WebSocket connection ID when a user logs in.

socket.on('TOKEN', (token) => {  
 document.cookie = `socket\_${socket.id}\_token=${token}; path=/;`;  
});

#### Read Token Based on Socket ID:

Extract the token specific to a socket ID.

function getToken(socketId) {  
 const cookie = document.cookie  
 .split('; ')  
 .find(row => row.startsWith(`socket\_${socketId}\_token=`));  
 return cookie ? cookie.split('=')[1] : null;  
}

#### Clean Up Cookies:

Remove cookies when a WebSocket connection is closed.

socket.on('close', () => {  
 document.cookie = `socket\_${socket.id}\_token=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;`;  
});

## Use Case 2: Varying DTO Sizes and Timeouts

### Problem:

* DTO sizes vary, making a fixed 30-second timeout ineffective.

### Solution:

#### Dynamic Timeout Based on DTO Size:

Calculate timeout dynamically based on DTO size.

int CalculateTimeout(UserDashboardDTO dto)  
{  
 int baseTimeout = 30; // Base timeout in seconds  
 int sizeFactor = dto.AllowedSections.Count + dto.AsyncScripts.Count + dto.SidebarOptions.Count;  
 return baseTimeout + (sizeFactor \* 2); // Add 2 seconds per item  
}

#### GUI Confirmation:

* GUI sends a DTO\_CONFIRMED message after processing the DTO.
* API waits for confirmation before proceeding.

#### Timeout Adjustment:

Use a sliding timeout that resets with progress updates.

var timeout = Task.Delay(TimeSpan.FromSeconds(CalculateTimeout(dto)));  
var completedTask = await Task.WhenAny(tcs.Task, timeout);

## Use Case 3: Cookie Management for Multiple Users

### Problem:

* Cookies are shared across users, leading to conflicts.

### Solution:

#### User-Specific Cookies:

Store tokens in cookies with a unique identifier.

document.cookie = `user\_${userId}\_token=${token}; path=/;`;

#### Associate Cookies with WebSocket Connections:

Map WebSocket connection IDs to user IDs.

const userSocketMap = new Map(); // userId -> socketId  
userSocketMap.set(userId, socket.id);

#### Read Token Based on User ID:

function getToken(userId) {  
 const cookie = document.cookie  
 .split('; ')  
 .find(row => row.startsWith(`user\_${userId}\_token=`));  
 return cookie ? cookie.split('=')[1] : null;  
}

#### Clean Up Cookies on Logout:

function logout(userId) {  
 document.cookie = `user\_${userId}\_token=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;`;  
 userSocketMap.delete(userId);  
}

## Use Case 4: Ensuring Token Integrity

### Problem:

* Tokens must be securely stored and protected from overwrites.

### Solution:

#### Encrypt Tokens in Cookies:

const encryptedToken = encrypt(token, secretKey);  
document.cookie = `user\_${userId}\_token=${encryptedToken}; path=/; HttpOnly; Secure; SameSite=Strict`;

#### Validate Tokens on the Server:

string DecryptToken(string encryptedToken)  
{  
 // Decryption logic  
 return decryptedToken;  
}

#### Use HttpOnly and Secure Flags:

* Prevents client-side scripts from accessing the token.
* Ensures cookies are only sent over HTTPS.

## Use Case 5: Handling Large DTOs

### Problem:

* Large DTOs may cause processing delays and timeouts.

### Solution:

#### Chunked DTO Transmission:

Split DTO into smaller chunks before sending.

var chunks = SplitDtoIntoChunks(dto, chunkSize);  
foreach (var chunk in chunks)  
{  
 socket.Send(chunk);  
}

#### Progress Updates:

GUI sends updates to track progress.

socket.emit('DTO\_PROGRESS', { chunkId: 1, status: 'processed' });

#### Dynamic Timeout Adjustment:

Adjust timeout based on chunk count and processing speed.

## Final Workflow

1. **User Logs In:**
   * API generates a token and sends it via WebSocket.
   * GUI stores the token in a user-specific cookie.
2. **Fetch DTO:**
   * GUI fetches and processes the DTO.
   * GUI sends a confirmation (DTO\_CONFIRMED) to the API.
3. **Timeout Handling:**
   * API waits for confirmation with a dynamic timeout.
4. **Logout:**
   * GUI removes the user-specific cookie and cleans up resources.

## Pros and Cons of the Proposed Solution

### Pros:

* **Scalable:** Handles multiple users and large DTOs efficiently.
* **Secure:** Encrypts tokens and uses HttpOnly cookies.
* **Flexible:** Implements dynamic timeouts and chunked DTO transmission.

### Cons:

* **Complexity:** Requires careful cookie and WebSocket management.
* **Performance Overhead:** Encryption and chunking introduce minor overhead.

## Future Improvements

* **Token Refresh:** Implement refresh tokens for long-lived sessions.
* **Load Balancing:** Distribute WebSocket connections across servers.
* **Monitoring:** Track WebSocket performance and cookie usage.