Secured Software Engineering
(COMSM0164)

Lab Exercise: Securing Real-Time APIs
(Part 2 – Server-Sent Events)

Prerequisite:
(1) Visual Studio Code (VS Code)
(2) A GitHub account
(3) Some knowledge of JavaScript and Node.js for API functional implementation.
(4) The client-side in this exercise also uses JavaScript, HTML and CSS. The latter two are optional prerequisite which enables a user-friendly and interactive web interface design.

Overview

Server Sent Events (SSE) is a protocol for enabling server initiated real-time message sending to clients. Unlike traditional RESTful APIs where requests and responses are initiated by the client to the server, unidirectional message passing is initiated from the server to the client. This exercise builds on the previous exercise on Implementing HTTPS APIs using SSL certificates. We will extend the implementation of E-Manager (an event management system) to explore how different APIs can interact in real-time using SSE, and the security implications associated with such interactions.

SSE can be implemented in two steps:
(1) The first is to implement a server-side resource endpoint that a client calls to start receiving events generated by that resource. The sample code is illustrated below:

```
const express = require('express');
const app = express();

app.get('/stocknotice', (req, res) => {
    res.setHeader('Content-Type', 'text/event-stream');
    res.setHeader('Cache-Control', 'no-cache');
    res.setHeader('Connection', 'keep-alive');

    // Function to send a message to the client when there is a new stock
    const sendMessage = (data) => {
        res.write(`${JSON.stringify(data)}\n\n`);
    };

    // Simulate sending a new stock message every 5 seconds
    var id = 1
    const interval = setInterval(() => {
        sendMessage({sid:id, stock:'item', timestamp:new
Date().toISOString()});
    }, 5000);
    // When the client closes the connection, stop sending stock messages
    req.on('close', () => {
        clearInterval(interval);
        res.end();
    });
});

app.listen(process.env.PORT_P2, () => {
    console.log(`Server running @ http://localhost:${process.env.PORT_P2}`);
});
```

In this example, the server listens for incoming connections on the /stocknotice endpoint and sends a stock message to the client every 5 seconds. When the client closes the connection, the server stops sending messages.

(2) The second is the client-side implementation for initialising a HTTP request to the resource endpoint. The sample code is illustrated below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <h1>Stock Ticker</h1>
    <div id="stockmessages"></div>
    <script>
        const eventSource = new EventSource('/stocknotice');

        eventSource.onmessage = function(event) {
            const message = JSON.parse(event.data);
            const messageElement = document.createElement('div');
            messageElement.textContent = `Stock:${message.timestamp}:
${message.message}`;
            document.getElementById('messages').appendChild(messageElement);
        };

        eventSource.onerror = function() {
            console.error('Error occurred in receipt of server-sent events.');
        };
    </script>
</body>
</html>
```

In this example, the client establishes an SSE connection to the server using the `EventSource` object. Messages received from the server are displayed in the `#stockmessages` div element.

Challenge Task

You will achieve the objective of this exercise by refactoring E-Manager's server-side implementation using Node.js and the client-side implementation using JavaScript. Specifically, extend the bearer authentication scheme (step 3) with the following features:

(1) Refactor E-Manager to automatically update all users on the platform each time a new event is added by a user. This new function will replace the `refresh()` function of the current implementation.

(2) Define a Cross-Origin Resource Sharing (CORS) Policy on the server API by configuring a strict CORS policy to control which domains are allowed to access your SSE endpoints.

(3) How can cross-site scripting (XSS) attacks be prevented

Additional Challenge Task

(4) Implement a rate limiting feature to prevent abuse and denial-of-service (DoS) attacks

(5) Would it be possible to Throttle the number of events sent to each client to avoid overwhelming them with too many updates.