Proyecto Intermodular

Programación de servicios y procesos

Acceso a Datos

Italo Ramirez Arellano DAM 2A

Índice

Desarrollo

Desarrollo del Cliente TCP

Clase StateObject: Esta clase se utiliza para almacenar el estado de la conexión y los datos recibidos

- WorkSocket: Un objeto Socket que representa el socket de trabajo.
- BufferSize: Constante que define el tamaño del búfer de lectura (1024 bytes).
- Buffer: Arreglo de bytes que actúa como buffer para los datos recibidos.
- Sb: Un StringBuilder para acumular los datos recibidos y construir mensajes completos.

Clase TokenResponse: Clase para deserializar la respuesta del login

- token: Cadena que almacena el token JWT recibido del servidor.
- Error: Cadena que almacena cualquier mensaje de error proporcionado por el servidor.

Clase HttpClientSocket: Esta es la clase principal que gestiona la comunicación del cliente con el servidor

- **Constantes y métodos auxiliares:**
  - Port: Define el puerto en el que el servidor escucha (8080 en este caso).
  - GetServerIpAddress(): Devuelve la dirección IP del servidor.
- **Método Main:**
  - Obtiene el token de autenticación llamando al método GetToken().
  - Si se recibe un token válido, procede a acceder a un recurso protegido utilizando el método AccessProtectedResource(token).
- **Método GetToken:**
  - Establece una conexión TCP con el servidor utilizando la dirección IP y el puerto definidos.
  - Prepara las credenciales de autenticación y las serializa en formato JSON.
  - Construye una solicitud HTTP POST para el endpoint /login, incluyendo las credenciales en el cuerpo de la solicitud.
  - Envía la solicitud al servidor y lee la respuesta.
  - Si la respuesta contiene un token válido, lo devuelve; de lo contrario, retorna una cadena vacía.
- **Método AccessProtectedResource:**
  - Establece una conexión TCP con el servidor utilizando la dirección IP y el puerto definidos.
  - Construye una solicitud HTTP GET para el endpoint /datos, incluyendo el token de autenticación en el encabezado Authorization.
  - Envía la solicitud al servidor y lee la respuesta, que representa el recurso protegido al que se intenta acceder.

Consideraciones adicionales:

- **Manejo de excepciones**: Cada método incluye bloques try-catch para capturar y manejar posibles excepciones que puedan ocurrir durante la ejecución, como errores de conexión o problemas al enviar/recibir los datos.
- **Codificación y decodificación**: El código utiliza Encoding.UTF8 para convertir cadenas a bytes y viceversa, asegurando que los datos se transmitan correctamente en formato UTF-8.
- **Serialización JSON**: Se emplea JsonSerializer para convertir objetos a formato JSON y para interpretar respuestas JSON del servidor, facilitando la comunicación estructurada entre el cliente y el servidor.
- **Conexiones TCP**: El uso de la clase Socket permite una gestión detallada de las conexiones TCP, proporcionando control sobre aspectos específicos de la comunicación en red.

Desarrollo del ServidorApiRest

Este servidor se encarga de que los clientes puedan realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sobre los datos de usuarios y citas.
La comunicación se efectúa mediante solicitudes HTTP (GET, POST, PUT, DELETE) y los datos se intercambian en formato JSON. La base de datos es un archivo local en formato JSON que, en una fase futura, se replicará en Firebase.

Estructura del Proyecto

El servidor API REST se construyó utilizando ASP.NET Core y se compone de varios elementos clave:

- **Modelos**: Representan la estructura de los datos.
    - **Usuario**: Define las propiedades de un usuario (Id, Nombre, Email, Teléfono) y contiene una lista de citas.
    - **Cita**: Define las propiedades de una cita (Id, Fecha, Servicio, Lugar, Método de Pago, FechaCreación).
- **Service**: Manejan la interacción con la base de datos en formato JSON.
    - **JsonDatabaseService**: Proporciona métodos para leer y guardar la lista de usuarios en un archivo database.json.
- **Controllers**: Gestionan las solicitudes HTTP y definen los endpoints del API.
    - **UsuariosController**: Expone las rutas para operar sobre usuarios y sobre la subcolección de citas asociadas a cada usuario.
- **Configuración**:
    - Se configuran los servicios necesarios como el servicio de base de datos y se definen las rutas para los controladores.
    - Los archivos appsettings.json y launchSettings.json configuran detalles como el logging, las URL de acceso y el entorno de desarrollo.

Securización

- **Autenticación y Autorización**
    - **Uso de JWT**:
      El API REST está diseñado para trabajar con autenticación basada en JSON Web Tokens (JWT).
      Los clientes deben enviar el token en el header Authorization en cada solicitud a los endpoints protegidos.
      El servidor verifica la validez del token antes de permitir el acceso a los recursos.
      (Actualmente, en la implementación básica, la validación puede estar simplificada y se planea mejorar la seguridad en futuras iteraciones.)
- **Registro de Operaciones CRUD**
    - **Logging**:
      Se registra cada operación CRUD (creación, actualización, eliminación) en un archivo de log.
      El registro incluye la fecha y hora de la operación, el tipo de operación, el ID del recurso afectado y la dirección IP del cliente que realizó la petición.
      Esto permite tener una trazabilidad y control de las acciones realizadas en el API.

Configuración y Despliegue

- **Program.cs**: Se configuran los servicios necesarios, por ejemplo, la inyección del JsonDatabaseService como un singleton, y se mapean los controladores a las rutas.
- **appsettings.json**: Se configuran los niveles de logging y otros parámetros del entorno.
- launchSettings.json:
- Define las URLs de lanzamiento y el entorno de desarrollo para pruebas locales.

Desarrollo del Servidor TCP

El **ServidorTCP** es el componente encargado de recibir y procesar las solicitudes de los clientes mediante conexiones TCP. Su función es actuar como intermediario en la comunicación, gestionando desde la autenticación (con JWT) hasta las operaciones CRUD sobre usuarios y citas. Además, implementa medidas de seguridad como cifrado asimétrico (RSA) y registra las operaciones realizadas en un archivo de log.

La comunicación se realiza mediante el protocolo HTTP sobre sockets, utilizando JSON para serializar y deserializar los datos. El servidor trabaja de forma asíncrona, lo que le permite atender múltiples conexiones simultáneamente sin bloquear el hilo principal.

Estructura del Proyecto

- Clase StateObject

  Esta clase se utiliza para mantener el estado de cada conexión.

  Campos principales:

  - workSocket: Almacena el objeto Socket de la conexión actual.
  - BufferSize: Tamaño del búfer (1024 bytes) para leer los datos.
  - buffer: Lista de bytes que se utiliza para almacenar temporalmente los datos recibidos.
  - sb: Un StringBuilder que acumula los datos recibidos hasta que se complete la petición.

```
// Clase para mantener el estado de la conexión
6 referencias
public class StateObject
{
    public Socket workSocket = null!;
    public const int BufferSize = 1024;
    public byte[] buffer = new byte[BufferSize];
    public StringBuilder sb = new StringBuilder();
}
```

- **Clase Credentials**:
  Representa las credenciales de login con dos propiedades:
  - Username
  - Password

```
// Modelo para representar las credenciales de login
1 referencia
public class Credentials
{
    2 referencias
    public string Username { get; set; } = string.Empty;
    1 referencia
    public string Password { get; set; } = string.Empty;
}
```

- **Clase Usuario:**
  Define la estructura de un usuario, con propiedades como:
  - Id (generado automáticamente con Guid.NewGuid().ToString())
  - Nombre
  - Email
  - Telefono

- Citas: Lista de objetos del tipo Cita

```
// Modelo para Usuario
12 referencias
public class Usuario
{
    5 referencias
    public string Id { get; set; } = Guid.NewGuid().ToString();
    2 referencias
    public string Nombre { get; set; } = string.Empty;
    2 referencias
    public string Email { get; set; } = string.Empty;
    2 referencias
    public string Telefono { get; set; } = string.Empty;
    4 referencias
    public List<Cita> Citas { get; set; } = new List<Cita>();
}
```

- **Clase Cita:**
  Define la estructura de una cita, con propiedades:
  - Id (generado automáticamente)
  - Fecha
  - Servicio
  - Lugar
  - MetodoPago
  - FechaCreacion (se asigna con DateTime.Now)

```
// Modelo para Cita
7 referencias
public class Cita
{
    4 referencias
    public string Id { get; set; } = Guid.NewGuid().ToString();
    2 referencias
    public DateTime Fecha { get; set; }
    2 referencias
    public string Servicio { get; set; } = string.Empty;
    2 referencias
    public string Lugar { get; set; } = string.Empty;
    2 referencias
    public string MetodoPago { get; set; } = string.Empty;
    0 referencias
    public DateTime FechaCreacion { get; set; } = DateTime.Now;
}
```

- **Clase JwtManager:**
  Gestiona la generación y validación de tokens JWT para la autenticación.
  - **Funcionalidades:**
    - GenerateToken(string username): Genera un token simple usando el nombre de usuario y la fecha actual.
    - ValidateToken(string token): Verifica si el token no es nulo ni vacío.

```
// Clase para gestionar los tokens JWT
2 referencias
public static class JwtManager
{
    private const string SecretKey = "mysupersecret_secret_key!1234567";
    private const int TokenValidityMinutes = 30;

    1 referencia
    public static string GenerateToken(string username)
    {
        // En este ejemplo se usa un método simple de generación de token.
        return Convert.ToBase64String(Encoding.UTF8.GetBytes(username + ":" + DateTime.UtcNow));
    }

    1 referencia
    public static bool ValidateToken(string token)
    {
        return !string.IsNullOrEmpty(token);
    }
}
```

- **Clase RSAEncryption:**
  Implementa el cifrado asimétrico utilizando RSA para asegurar la comunicación.
  - **Métodos clave:**
    - Initialize(): Genera un par de claves RSA (2048 bits).
    - GetPublicKeyString(): Retorna la clave pública en formato Base64, para que el cliente la use al encriptar mensajes.
    - Decrypt(string base64Encrypted): Desencripta un mensaje recibido (esperado en Base64).
    - Encrypt(string message, string base64PublicKey): (Método opcional) Encripta un mensaje usando una clave pública.

```
// Clase para implementar cifrado asimétrico con RSA
3 referencias
public static class RSAEncryption
{
    private static RSA? rsa;

    // Inicializa la clave RSA (se genera un par de claves)
    1 referencia
    public static void Initialize()
    {
        rsa = RSA.Create(2048);
    }

    // Devuelve la clave pública en formato Base64 (para que el cliente la use para encriptar)
    1 referencia
    public static string GetPublicKeyString()
    {
        if (rsa == null)
            throw new Exception("RSA no inicializado.");
        // Exporta la clave pública en formato PEM
        var publicKey = rsa.ExportSubjectPublicKeyInfo();
        return Convert.ToBase64String(publicKey);
    }

    // Desencripta el mensaje encriptado (se espera que el mensaje esté en Base64)
    1 referencia
    public static string Decrypt(string base64Encrypted)
    {
        if (rsa == null)
            throw new Exception("RSA no inicializado.");
        byte[] encryptedBytes = Convert.FromBase64String(base64Encrypted);
        byte[] decryptedBytes = rsa.Decrypt(encryptedBytes, RSAEncryptionPadding.Pkcs1);
        return Encoding.UTF8.GetString(decryptedBytes);
    }
```

```
// Método opcional para encriptar (usado en el lado del cliente)
0 referencias
public static string Encrypt(string message, string base64PublicKey)
{
    byte[] publicKeyBytes = Convert.FromBase64String(base64PublicKey);
    using RSA rsaPublic = RSA.Create();
    rsaPublic.ImportSubjectPublicKeyInfo(publicKeyBytes, out _);
    byte[] messageBytes = Encoding.UTF8.GetBytes(message);
    byte[] encryptedBytes = rsaPublic.Encrypt(messageBytes, RSAEncryptionPadding.Pkcs1);
    return Convert.ToBase64String(encryptedBytes);
}
```

Clase HttpServer (Servidor TCP)

Esta es la clase principal del servidor y del proyecto, y contiene la lógica para establecer, recibir y procesar las conexiones de los clientes.

Método Main

- **Función:**
    - Inicializa el cifrado asimétrico llamando a RSAEncryption.Initialize().
    - Imprime en la consola la clave pública (para que el cliente pueda usarla para encriptar).
    - Llama a StartListening() para iniciar el proceso de escucha de conexiones.
- **Importancia:**
  Es el punto de entrada del programa, asegurando que antes de aceptar conexiones, se configure el cifrado.

```
public static void Main(string[] args)
{
    // Generamos el par de claves RSA al iniciar el servidor (se usarán en toda la sesión)
    RSAEncryption.Initialize();
    Console.WriteLine("Clave pública del servidor (envíala al cliente para encriptar):");
    Console.WriteLine(RSAEncryption.GetPublicKeyString());
    StartListening();
}
```

Método StartListening()

- **Función:**
    - Obtiene la dirección IP local mediante GetLocalIpAddress().
    - Crea un IPEndPoint utilizando la IP y el puerto (8080).
    - Crea un socket listener y lo enlaza al endpoint.
    - Comienza a escuchar hasta 100 conexiones simultáneas.
    - Entra en un bucle infinito en el que espera conexiones:
        - Reinicia el ManualResetEvent con allDone.Reset().
        - Llama a BeginAccept para aceptar una nueva conexión de forma asíncrona.
        - Usa allDone.WaitOne() para pausar hasta que se acepte una conexión.
- **Importancia:**
  Establece el servidor para recibir conexiones de manera asíncrona y concurrente.

```
public static void StartListening()
{
    IPAddress ipAddress = GetLocalIpAddress();
    IPEndPoint localEndPoint = new IPEndPoint(ipAddress, PORT);
    Socket listener = new Socket(ipAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

    try
    {
        listener.Bind(localEndPoint);
        listener.Listen(100);
        Console.WriteLine("Servidor HTTP escuchando en {0}:{1}", ipAddress, PORT);

        while (true)
        {
            allDone.Reset();
            Console.WriteLine("Esperando conexión...");
            listener.BeginAccept(new AsyncCallback(AcceptCallback), listener);
            allDone.WaitOne();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: {0}", ex.ToString());
    }
}
```

Método AcceptCallback(IAsyncResult ar)

- **Función:**
  - Se ejecuta cuando se acepta una conexión.
  - Llama a EndAccept para obtener el socket del cliente.
  - Crea un objeto StateObject y asigna el socket de trabajo.
  - Llama a BeginReceive para empezar a recibir datos del cliente de forma asíncrona.
- **Importancia:**
  Maneja la aceptación de nuevas conexiones y prepara el objeto de estado para recibir datos.

```
1 referencia
public static void AcceptCallback(IAsyncResult ar)
{
    allDone.Set();
    Socket listener = (Socket)ar.AsyncState!;
    Socket handler = listener.EndAccept(ar);

    StateObject state = new StateObject();
    state.workSocket = handler;
    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
        new AsyncCallback(ReadCallback), state);
}
```

Método ReadCallback(IAsyncResult ar)

- **Función:**
  - Se invoca cada vez que se reciben datos de un cliente.
  - Utiliza el StateObject para acumular datos en el StringBuilder.
  - Cuando detecta que la petición HTTP está completa (contiene "\r\n\r\n"), procesa la petición.
  - **Cifrado:**
    - Si el mensaje empieza con "ENCRYPTED:", se extrae y se desencripta usando RSAEncryption.Decrypt().
  - **Ruteo:**
    - Extrae la primera línea de la petición para obtener el método (GET, POST, etc.) y la ruta.
    - Según la ruta:
      - Si es /login y método POST: procesa la autenticación.
      - Si es /datos y método GET: verifica el token JWT y da acceso al recurso.
      - Si la ruta empieza con /usuarios: llama a HandleCrudOperations() para gestionar operaciones CRUD.
  - Finalmente, envía la respuesta al cliente usando SendResponse().
- **Importancia:**
  Es el corazón del servidor, donde se interpreta y procesa la lógica de negocio según la solicitud recibida.

Método SendResponse(Socket handler, string httpResponse)

- **Función:**
  - Convierte la respuesta HTTP (cadena) a bytes.
  - Utiliza BeginSend para enviar los datos de forma asíncrona al cliente.
- **Importancia:**
  Se encarga de enviar la respuesta al cliente de manera no bloqueante.

```
private static void SendResponse(Socket handler, string httpResponse)
{
    byte[] byteData = Encoding.ASCII.GetBytes(httpResponse);
    handler.BeginSend(byteData, 0, byteData.Length, 0,
        new AsyncCallback(SendCallback), handler);
}
```

Método SendCallback(IAsyncResult ar)

- **Función:**
  - Finaliza el proceso de envío de datos usando EndSend.
  - Imprime en la consola la cantidad de bytes enviados.
  - Realiza el cierre de la conexión con Shutdown y Close.

- **Importancia:**
  Asegura que la conexión se cierre correctamente después de enviar la respuesta.

```
1 referencia
private static void SendCallback(IAsyncResult ar)
{
    try
    {
        Socket handler = (Socket)ar.AsyncState!;
        int bytesSent = handler.EndSend(ar);
        Console.WriteLine("Se enviaron {0} bytes al cliente.", bytesSent);
        handler.Shutdown(SocketShutdown.Both);
        handler.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error en SendCallback: {0}", ex.ToString());
    }
}
```

Método GetJwtFromRequest(string request)

- Función:
  - Recorre las líneas de la petición HTTP para encontrar el header Authorization.
  - Extrae y retorna el token JWT presente en el header.
- Importancia:
  Permite validar que el cliente ha enviado el token necesario para acceder a recursos protegidos.

```
// Extrae el token JWT del header Authorization de la petición HTTP
1 referencia
private static string GetJwtFromRequest(string request)
{
    string[] lines = request.Split(new string[] { "\r\n" }, StringSplitOptions.None);
    foreach (string line in lines)
    {
        if (line.StartsWith("Authorization:", StringComparison.InvariantCultureIgnoreCase))
        {
            string[] parts = line.Split(' ', StringSplitOptions.RemoveEmptyEntries);
            if (parts.Length >= 3 && parts[1].Trim() == "Bearer")
            {
                return parts[2].Trim();
            }
        }
    }
    return "";
}
```

Método GetLocalIpAddress()

- **Función:**
  - Obtiene todas las direcciones IP asociadas al nombre del host.
  - Retorna la primera dirección IPv4 que no sea la de loopback (127.0.0.1).
- **Importancia:**
  Proporciona la dirección IP local válida para enlazar el socket del servidor.

```
// Obtiene una dirección IP local válida (si no encuentra otra, retorna Loopback)
1 referencia
private static IPAddress GetLocalIpAddress()
{
    IPHostEntry host = Dns.GetHostEntry(Dns.GetHostName());
    foreach (IPAddress ip in host.AddressList)
    {
        if (ip.AddressFamily == AddressFamily.InterNetwork && !ip.Equals(IPAddress.Loopback))
        {
            return ip;
        }
    }
    return IPAddress.Loopback;
}
```

Método HandleCrudOperations(string method, string path, string body, string clientInfo)

- **Función:**
  - Gestiona todas las operaciones CRUD sobre usuarios y citas.
  - Separa la ruta en segmentos para determinar si la petición afecta a usuarios o a citas.
  - Según el método HTTP:
    - **Usuarios:**
      - GET: Retorna la lista de usuarios (con un retraso simulado de 5 segundos para probar la concurrencia).
      - POST: Crea un nuevo usuario a partir del JSON recibido.
      - PUT: Actualiza un usuario existente.
      - DELETE: Elimina un usuario.
    - **Citas:**
      - Para rutas anidadas bajo /usuarios/{id}/citas, permite crear, leer, actualizar y eliminar citas asociadas a un usuario.
  - Cada operación CRUD registra la actividad en un archivo de log mediante LogCrudOperation().
- **Importancia:**
  Centraliza la lógica de negocio para gestionar los datos y permite mantener un registro de las operaciones realizadas.

Métodos LoadUsuarios() y SaveUsuarios(List<Usuario> usuarios)

- **Función:**
  - LoadUsuarios(): Lee el contenido del archivo database.json y deserializa la lista de usuarios.
  - SaveUsuarios(): Serializa la lista de usuarios en formato JSON y la guarda en el archivo database.json.
- **Importancia:**
  Permiten la persistencia de los datos de manera simple utilizando un archivo JSON como base de datos local.

```
// Métodos para cargar y guardar los usuarios en el archivo JSON
1 referencia
private static List<Usuario> LoadUsuarios()
{
    string filePath = "database.json";
    if (!File.Exists(filePath))
        return new List<Usuario>();
    string json = File.ReadAllText(filePath);
    return JsonSerializer.Deserialize<List<Usuario>>(json) ?? new List<Usuario>();
}

6 referencias
private static void SaveUsuarios(List<Usuario> usuarios)
{
    string filePath = "database.json";
    string json = JsonSerializer.Serialize(usuarios, new JsonSerializerOptions { WriteIndented = true });
    File.WriteAllText(filePath, json);
}
```

Método LogCrudOperation(string operation, string resourceId, string clientInfo)

- **Función:**
  - Escribe en un archivo de log (crud_log.txt) una línea con la fecha, la operación realizada, el ID del recurso afectado y la información del cliente (por ejemplo, la dirección IP).
- **Importancia:**
  Proporciona trazabilidad y auditoría de las operaciones CRUD realizadas en el sistema.

```
// Función para registrar las operaciones CRUD en un archivo de log
6 referencias
private static void LogCrudOperation(string operation, string resourceId, string clientInfo)
{
    string logEntry = $"{DateTime.Now}: Operación {operation} sobre el recurso {resourceId} realizada por {clientInfo}";
    File.AppendAllText("crud_log.txt", logEntry + Environment.NewLine);
}
```

Conclusiones

Este proyecto ha sido de mucho aprendizaje. Al principio, me sentía bastante perdido, no sabía por dónde empezar, decidí empezar por el cliente-servidor que es de lo que más sabía, pude lograrlo con los ejemplos y buscando información. Luego implementar la autenticación con JWT y usar RSA para encriptar los mensajes me hizo pensar en lo importante que es implementar securización en cualquier aplicación, sobre todo sin son datos muy valiosos, además esto no era tan complejo como yo pensaba.

Además, desarrollar el API REST con ASP.NET Core y trabajar con JSON para enviar y recibir datos fue algo bastante práctico y que ya entendía el concepto pero en Java, en C# es bastante parecido pero diferente a la vez (por el lenguaje). Pude ver en tiempo real cómo se crean, leen, actualizan y eliminan usuarios y citas, lo que me ayudó a comprender mejor cómo funciona.

Aunque todavía hay áreas por mejorar y otras que no terminó muy bien de entender, este proyecto me permitió combinar varios conceptos y técnicas que me preparan para enfrentar retos aún mayores. Lo importante es que dividiendo el problema, se pueden resolver las cosas.

Para finalizar, algo más personal, me esforcé mucho en este proyecto y estoy satisfecho con el resultado. Ver que las cosas funcionan es una satisfacción única que he probado desde que empecé a estudiar esto y aunque no todo fue bueno para mí, creo que puedo salir adelante ;).

Deudas técnicas

Implementación de Firebase:

Aunque se planea replicar los datos en Firebase, por el momento decidí que no sea así, lo que limita la persistencia a largo plazo y la escalabilidad en la nube.

Gestión simplificada del JWT:

La generación y validación de los tokens JWT se hace de forma muy básica. Para un entorno de producción sería ideal usar una librería especializada (como System.IdentityModel.Tokens.Jwt) para una validación y control de expiración más robustos.

Código no dividido:

Actualmente gran parte del código se encuentra en un solo archivo (Program.cs del Servidor TCP), lo que dificulta la mantenibilidad y escalabilidad. Sería conveniente separar las clases en archivos.

Pruebas y validación:

No se han implementado pruebas unitarias o de integración que permitan asegurar que cada componente funcione correctamente en diferentes escenarios, solo manuales con Postman.