Documentación Gestión Reservas Restaurante

1. Introducción

Este proyecto implementa un sistema de reservas con una arquitectura distribuida, que incluye una API REST, un servidor intermedio y clientes que se comunican mediante sockets seguros.

---

2. Arquitectura del Sistema

El sistema se compone de los siguientes módulos:

- **API REST**: Gestiona la lógica de negocio y el acceso a la base de datos.
- **Servidor Intermedio**: Actúa como intermediario entre los clientes y la API.
- **Clientes**: Aplicaciones que interactúan con el servidor para realizar reservas.
- **Seguridad**: Comunicación cifrada y autenticación mediante firmas digitales.

---

3. API REST

La API REST se implementa en .NET y expone endpoints para gestionar las reservas.

Endpoints principales

- `POST /reservas` → Crear una reserva.
- `GET /reservas/{id}` → Obtener detalles de una reserva.
- `PUT /reservas/{id}` → Actualizar una reserva.
- `DELETE /reservas/{id}` → Cancelar una reserva.
- `GET /reservas` → Listar todas las reservas.

---

4. Servidor Intermedio

Este servidor actúa como un proxy seguro entre los clientes y la API.

- **Autenticación** de clientes mediante claves RSA.
- **Cifrado de comunicaciones** con claves públicas y privadas.
- **Auditoría** de todas las operaciones.

---

5. Cliente

Cliente de API (`ApiClient`)

La clase `ApiClient` maneja la comunicación con la API REST.

```
public class ApiClient
{
    private readonly HttpClient _httpClient;
    private readonly string _baseUrl;

    public ApiClient(string baseUrl)
    {
        _baseUrl = baseUrl;
        _httpClient = new HttpClient();
    }

    public async Task<string> SendRequestAsync(string action, string
data)
```

```
    {
        try
        {
            var (operation, endpoint) = ParseAction(action);

            HttpResponseMessage response = operation switch
            {
                CrudOperation.Create => await
_httpClient.PostAsync($"{_baseUrl}/{endpoint}",
                    new StringContent(data, Encoding.UTF8,
"application/json")),
                CrudOperation.Read => await
_httpClient.GetAsync($"{_baseUrl}/{endpoint}"),
                CrudOperation.Update => await
_httpClient.PutAsync($"{_baseUrl}/{endpoint}",
                    new StringContent(data, Encoding.UTF8,
"application/json")),
                CrudOperation.Delete => await
_httpClient.DeleteAsync($"{_baseUrl}/{endpoint}"),
                CrudOperation.List => await
_httpClient.GetAsync($"{_baseUrl}/{endpoint}"),
                _ => throw new NotSupportedException($"Operación no
soportada: {operation}")
            };

            response.EnsureSuccessStatusCode();
            return await response.Content.ReadAsStringAsync();
        }
        catch (Exception ex)
        {
            return JsonConvert.SerializeObject(new { error =
ex.Message });
        }
    }
}
```

---

6. Script de Inicio (`IniciarAplicacion.ps1`)

El script en PowerShell facilita el inicio de todos los componentes:

```
$scriptDir = $PSScriptRoot

$apiPath = Join-Path -Path $scriptDir -ChildPath "API"
$intermediateServerPath = Join-Path -Path $scriptDir -ChildPath
"IntermediateServer"
$clientPath = Join-Path -Path $scriptDir -ChildPath "Clients\Client"

$apiPort = 5138
$serverPort = 8080

function Start-Api {
    Write-Host "Iniciando API REST..." -ForegroundColor Cyan
    Set-Location -Path $apiPath
    Start-Process -FilePath "dotnet" -ArgumentList "run"
-NoNewWindow
}

function Start-IntermediateServer {
    Write-Host "Iniciando Servidor Intermedio..." -ForegroundColor
Green
    Set-Location -Path $intermediateServerPath
    Start-Process -FilePath "dotnet" -ArgumentList "run"
-NoNewWindow
}

function Start-Client {
    Write-Host "Iniciando Cliente..." -ForegroundColor Yellow
    Start-Process -FilePath "dotnet" -ArgumentList "run"
-WorkingDirectory $clientPath
}

Start-Api
Start-Sleep -Seconds 5
Start-IntermediateServer
Start-Sleep -Seconds 3
Start-Client
```

---

7. Aspectos Técnicos Destacados

Comunicación Asíncrona con Sockets

El proyecto implementa comunicación asíncrona mediante sockets TCP, permitiendo múltiples conexiones simultáneas:

```
// En SocketServer.cs
private async Task AcceptConnectionsAsync()
{
    while (_isRunning)
    {
        Socket clientSocket = await _serverSocket.AcceptAsync();
        string clientId = Guid.NewGuid().ToString();

        var clientConnection = new ClientConnection(clientId,
clientSocket, _cryptoService);
        _connectedClients.TryAdd(clientId, clientConnection);

        // Iniciar el procesamiento de mensajes para este cliente en
una tarea separada
        _ = ProcessClientMessagesAsync(clientConnection);
    }
}
```

Cifrado Asimétrico RSA

El sistema implementa cifrado asimétrico RSA para proteger la confidencialidad de las comunicaciones:

```
// En CryptoServiceBase.cs
public byte[] Encrypt(byte[] data, RSAParameters publicKey)
{
    using (var rsa = RSA.Create())
    {
        rsa.ImportParameters(publicKey);
        return rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA256);
    }
}

public byte[] Decrypt(byte[] encryptedData)
{
    return _privateKey.Decrypt(encryptedData,
RSAEncryptionPadding.OaepSHA256);
```

```
}
```

Firmas Digitales

El sistema utiliza firmas digitales para garantizar la autenticidad e integridad de los mensajes:

```
// En CryptoServiceBase.cs
public byte[] SignData(byte[] data)
{
    return _privateKey.SignData(data, HashAlgorithmName.SHA256,
RSASignaturePadding.Pkcs1);
}

public bool VerifySignature(byte[] data, byte[] signature,
RSAParameters publicKey)
{
    using (var rsa = RSA.Create())
    {
        rsa.ImportParameters(publicKey);
        return rsa.VerifyData(data, signature,
HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    }
}
```

Control de Concurrencia

El sistema utiliza semáforos para proteger el acceso a recursos compartidos:

```
// En JsonDataService.cs
private readonly SemaphoreSlim _semaphore = new SemaphoreSlim(1, 1);

public async Task<Reservation> CreateReservationAsync(Reservation
reservation)
{
    await _semaphore.WaitAsync();
    try
    {
        // Asignar un ID único
        reservation.Id = DateTime.Now.Ticks;
        reservation.CreatedAt = DateTime.Now;
```

```
            // Añadir la nueva reserva a la lista
            _reservations.Add(reservation);

            // Guardar los cambios en el archivo JSON
            await SaveReservationsAsync();

            return reservation;
        }
        finally
        {
            _semaphore.Release();
        }
    }
```

---

8. Requisitos Cumplidos

RA3 - Programación de comunicaciones en red:

Implementación de sockets TCP para comunicación cliente-servidor.
Gestión asíncrona de múltiples conexiones simultáneas.
Serialización y deserialización de mensajes en formato JSON.
Control de concurrencia mediante semáforos.

RA4 - Generación de servicios en red:

API REST completa con operaciones CRUD.
Persistencia de datos en archivos JSON.
Servidor intermedio que actúa como proxy.
Documentación de API con Swagger.

RA5 - Técnicas criptográficas y programación segura:

Cifrado asimétrico RSA para proteger las comunicaciones.
Firmas digitales para autenticar el origen de los mensajes.
Registro de auditoría para todas las operaciones.
Identificación y registro de usuarios que realizan operaciones.

---

9. Conclusiones y Aprendizajes

Este proyecto ha permitido aplicar conceptos avanzados de programación de servicios y procesos, desarrollando un sistema robusto y seguro para la gestión de reservas. 🚀