Sultan Qaboos University
College of Science
Department of Computer Science
COMP4701 – Web Application Development

Project

Intrusion Detection System

Submitted to: Dr. Abdullah Al-Hamdani

Team Members:
Amr Al Maashani - 135261
Osama Al Busaidi - 134814
Mojahid Al Souti - 132860

Date: 10/11/2024

Table of Contents

Project Specifications

Project Information

Title

Intrusion Detection System (IDS) Platform.

Objectives

- To create a responsive web application with real-time monitoring and intrusion detection features.

- To put algorithms in place that can identify irregularities in network traffic and notify users of questionable activity.

- To provide consumers with a data-driven, visual interface to assess and averting possible risks.

Motivations

- Real-time intrusion detection is crucial for protecting sensitive data due to the growing complexity of cyber threats. Intrusion detection systems (IDS) is a useful mechanism to current security configurations as traditional security measures frequently fail to identify complex threats.

- With an easy-to-use interface, the online application seeks to enable small and medium-sized businesses to monitor and defend their networks. By offering a comprehensive and easy to read dashboard, this initiative improves accessibility and emphasizes the significance of real-time monitoring.

Why This Topic?
Real-World Impact

- Globally, cybersecurity risks are becoming a bigger worry for both people and businesses. By identifying possible intrusions, an IDS gives users the ability to react quickly, adding an extra layer of protection. Any firm trying to protect its networks from cyberattacks, illegal access, and data breaches should consider this product.

- **User Base**: Small and medium businesses (SMEs), IT security teams, and individual users with networks to keep an eye on are the target audience.

Project Requirements

Functional Requirements (Main Features)

- **User authentication and authorization**: a safe login process with varying degrees of access (e.g., normal user, admin).
- **Intrusion Detection Dashboard**: This feature shows network traffic in real time and provides notifications of questionable activity.
- **Alert System**: Notifications of intrusions found, together with information on their severity and recommended courses of action.
- **Traffic Analysis**: Records and examines network data to find trends and anomalies.
- **Report Generation**: Create and obtain reports on network status and issues found over certain time frames.

Non-Functional Requirements

- **Performance**: Minimal delay in real-time reaction to detected intrusions.
- **Scalability**: As the user base expands, accommodate higher data loads and more network traffic.
- **Usability**: A user-friendly interface that is usable by people with different levels of technological expertise.
- **Security** : include tight access control and the encryption of private information.
- **Compatibility**: Verify that the application works properly across the majority of web browsers.

Project Design

Architecture Overview

1. **Front-End:** Developed using React, it provides an interactive user interface where users can monitor network activities, view alerts, and manage settings. Key pages include a **Dashboard**, **Traffic Logs**, **Alerts**, **Reports**, and **User Settings**.
2. **Back-End:** ASP.NET Core handles the main application logic, manages user authentication, processes data, and provides APIs for the front-end to fetch data.

Information Flow

- **User Login:** User logs in via the React front end, and authentication is processed by ASP.NET Core.

- **Data Collection and Detection:** Network data is processed in backend in real time, analyzed for threats, and results are communicated via ASP.NET Core.

- **Data Presentation:** ASP.NET Core retrieves processed data and transmits it to React for display in the user's dashboard.

- **Alert and Notification System:** Alerts are generated in Backend and routed throgh ASP.NET Core to be displayed in React, allowing for real-time notifications.

Database Design

This is only the main idea (there are deeper classes that is not feasible to show in an ERD)

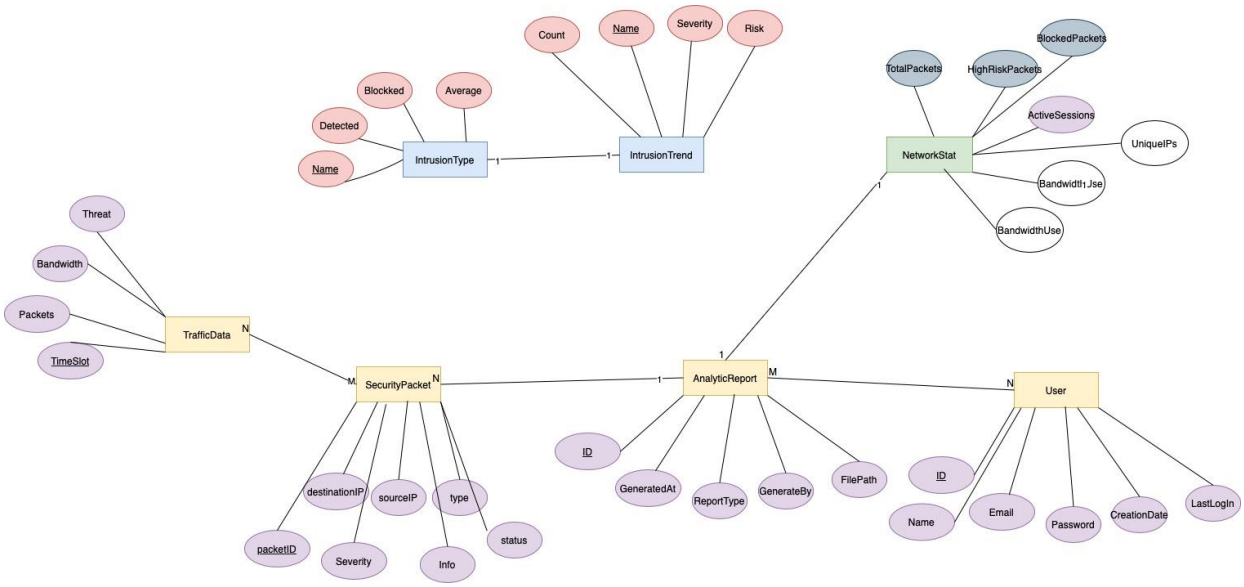


Figure 1 ERD for database tables

User

Purpose: Stores essential user information for access control and tracking.

- **Attributes**:

- Id (PK)
- FullName
- Email
- PasswordHash
- CreatedAt
- LastLoginAt

IntrusionTrend

Purpose: Logs trends related to detected and blocked intrusions.

- **Attributes**:
  - Name (PK)
  - Detected
  - Blocked
  - Average

IntrusionType

Purpose: Defines types of intrusions with associated severity and risk levels.

- **Attributes**:
  - Name (PK)
  - Count
  - Severity
  - Risk

NetworkStats

Purpose: Tracks network performance statistics.

- **Attributes**:
  - TotalPackets
  - HighRiskPackets
  - ActiveSessions
  - BandwidthUsage
  - UniqueIPs
  - ThreatsBlocked
  - Changes (key-value pairs, may need a separate table if expanded)

TrafficData

Purpose: Logs traffic data with time slots.

- **Attributes**:
  - TimeSlot (PK)

- Packets
- Bandwidth
- Threats

ProtocolDistribution

Purpose: Stores information about the distribution of network protocols in the monitored traffic.

- **Attributes**:
  - Name (PK)
  - Value

SecurityPacket

Purpose: Represents packets flagged by IDS as security concerns.

- **Attributes**:
  - Id (PK)
  - IpSrc
  - IpDst
  - Info
  - Type
  - Status
  - Severity
  - Timestamp

AnalyticsReport

Purpose: Stores analytics report data and includes detailed network statistics and associated packets.

- **Attributes**:
  - Id (PK)
  - GeneratedAt
  - ReportType
  - GeneratedBy
  - FilePath
- **Relationships**:
  - Stats: 1-to-1 relationship with NetworkStats
  - Packets: 1-to-Many relationship with SecurityPacket

Relationships

Relationships are temporary for now and may change later based on the project context.

- **User ↔ AnalyticsReport**: A 1-to-Many relationship where each user may generate multiple analytics reports.

- **AnalyticsReport ↔ NetworkStats**: A 1-to-1 relationship where each report has one associated network statistic entry.
- **AnalyticsReport ↔ SecurityPacket**: A 1-to-Many relationship where each report may contain multiple security packets.
- **SecurityPacket ↔ IntrusionType**: A Many-to-1 relationship, linking each security packet with a specific intrusion type for categorization.

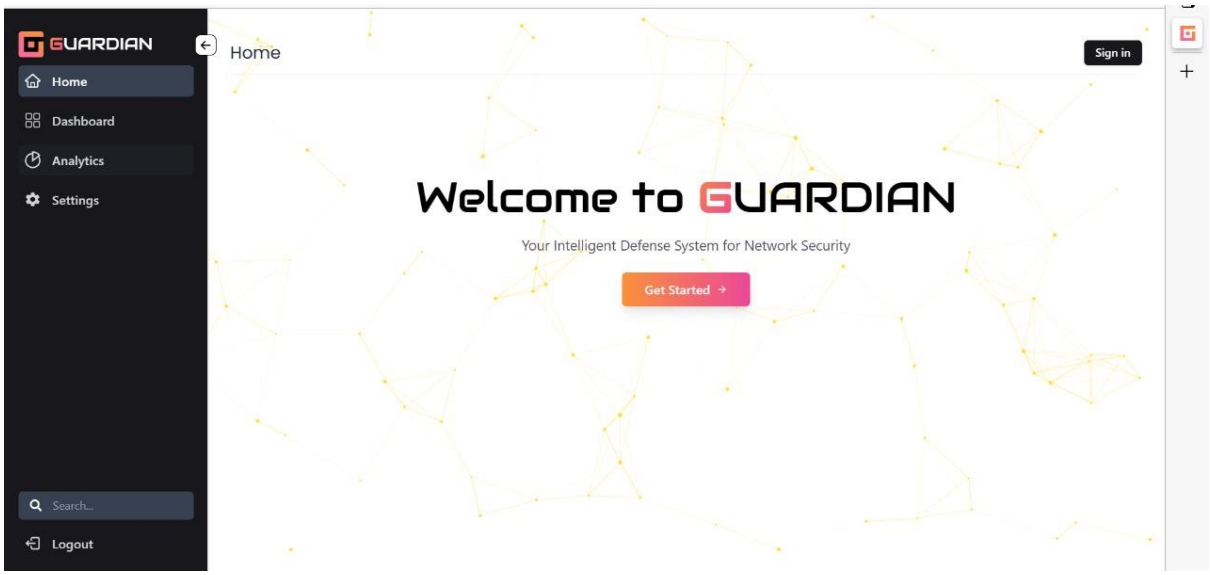
Design

This project's design combines React on the frontend with ASP.NET on the backend to produce a responsive, user-friendly application.
In addition to managing all server-side functionality, data processing, and safe data storage, ASP.NET offers a collection of API endpoints that interface with the frontend with ease.

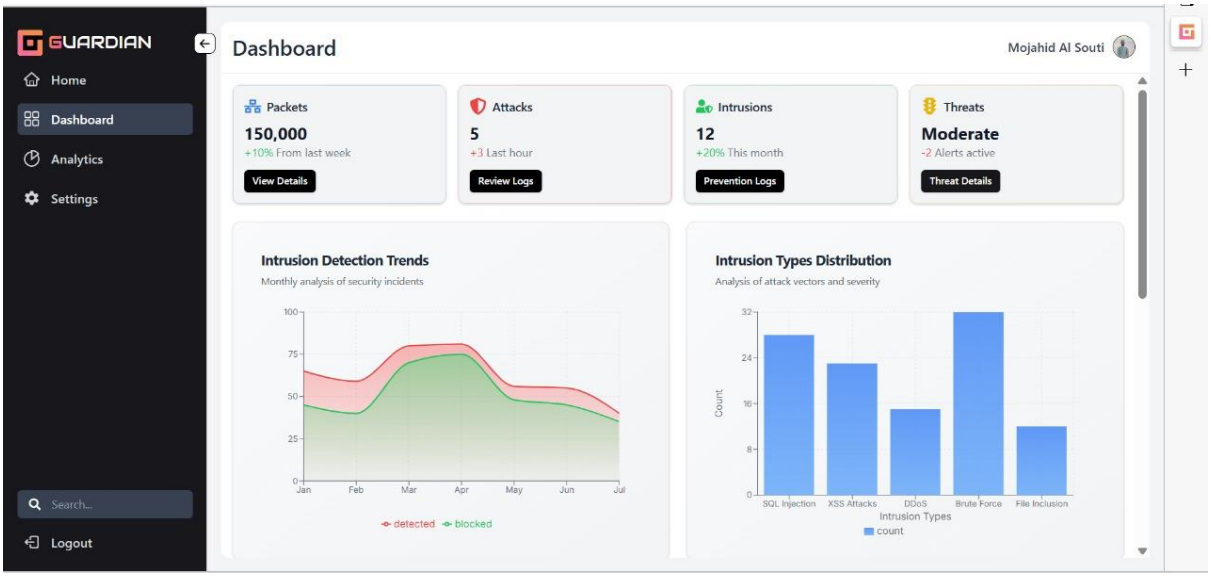
Real-time interactions are made possible by this API layer, which makes sure that data flows easily between the client and server. We can create reusable user interface components on the frontend thanks to React's component-based architecture, which not only speeds up development but also keeps the code efficient and manageable.

Throughout, the priority has been on developing a responsive, seamless experience with a simple interface that people find quick and easy to use. Additionally, this design strategy allows for future expansion, enabling us to scale and add features gradually without requiring significant redesigns.

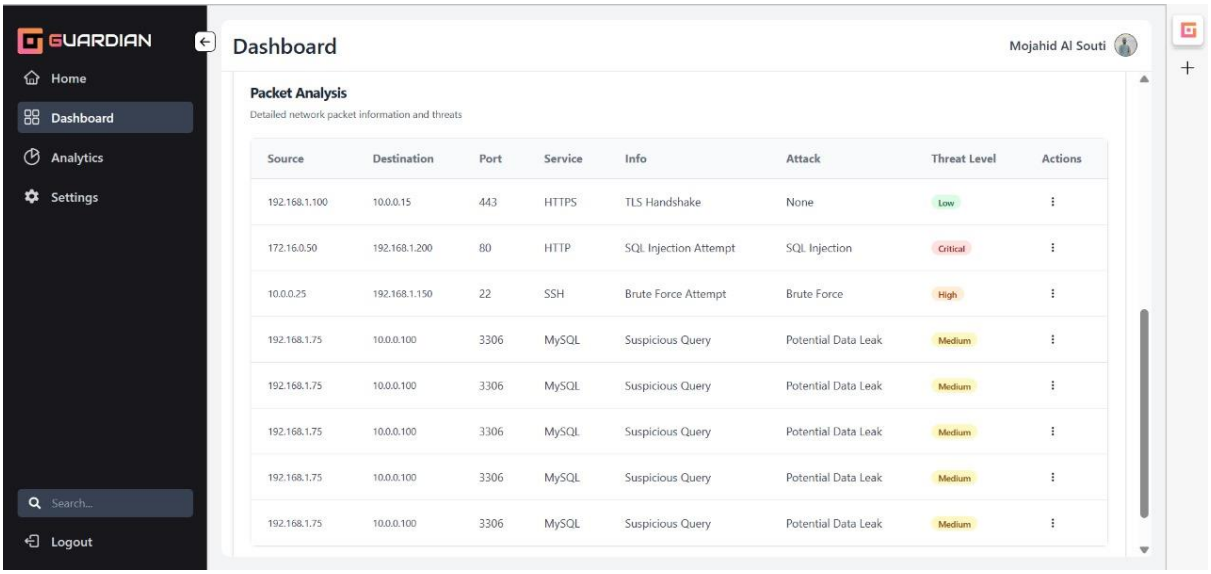
Screenshots of the web application:



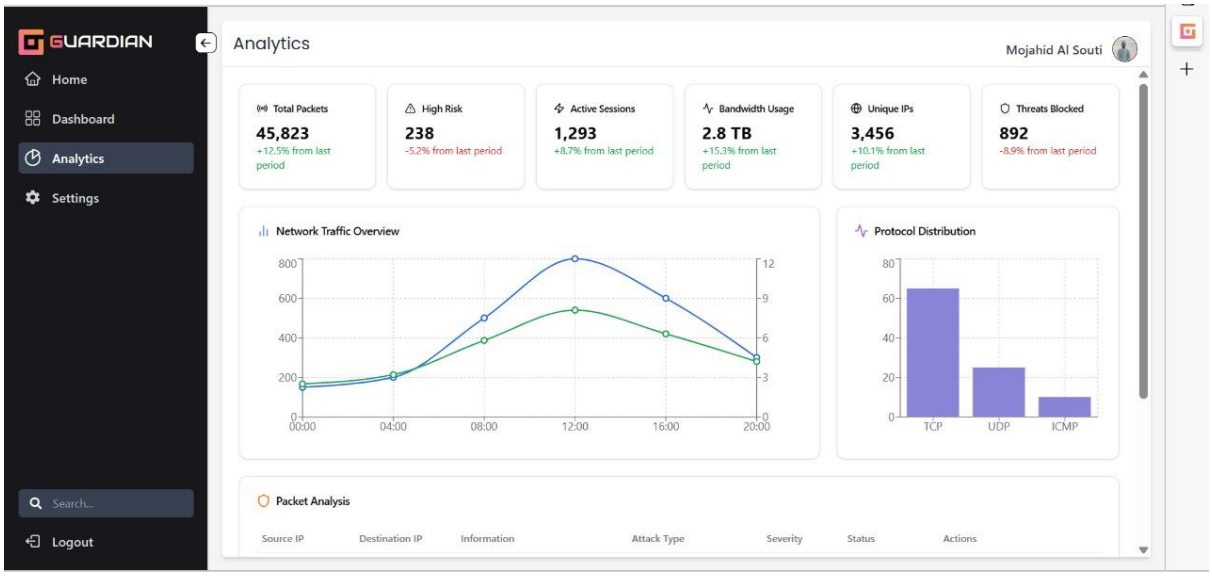
Picture 1 Home Page



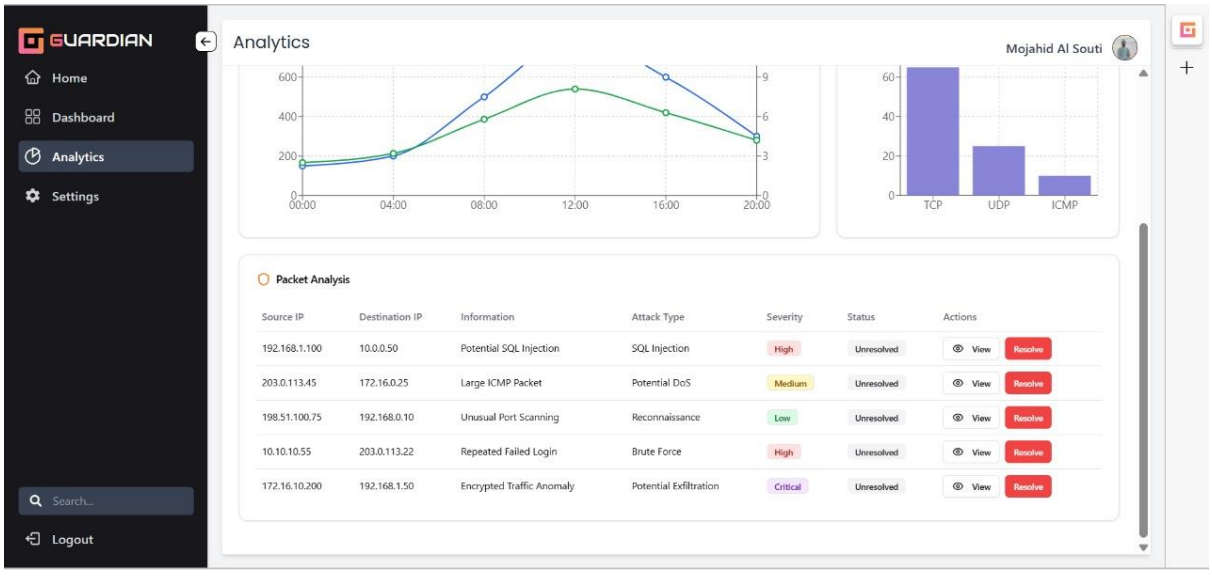
Picture 2 Dashboard



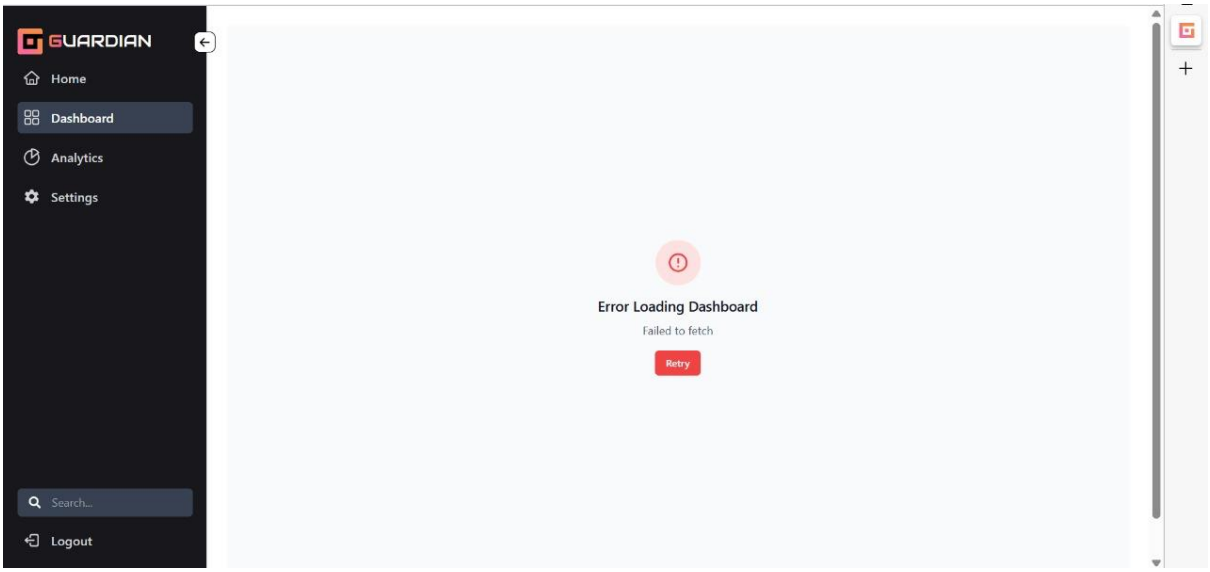
Picture 3 Dashboard contd.



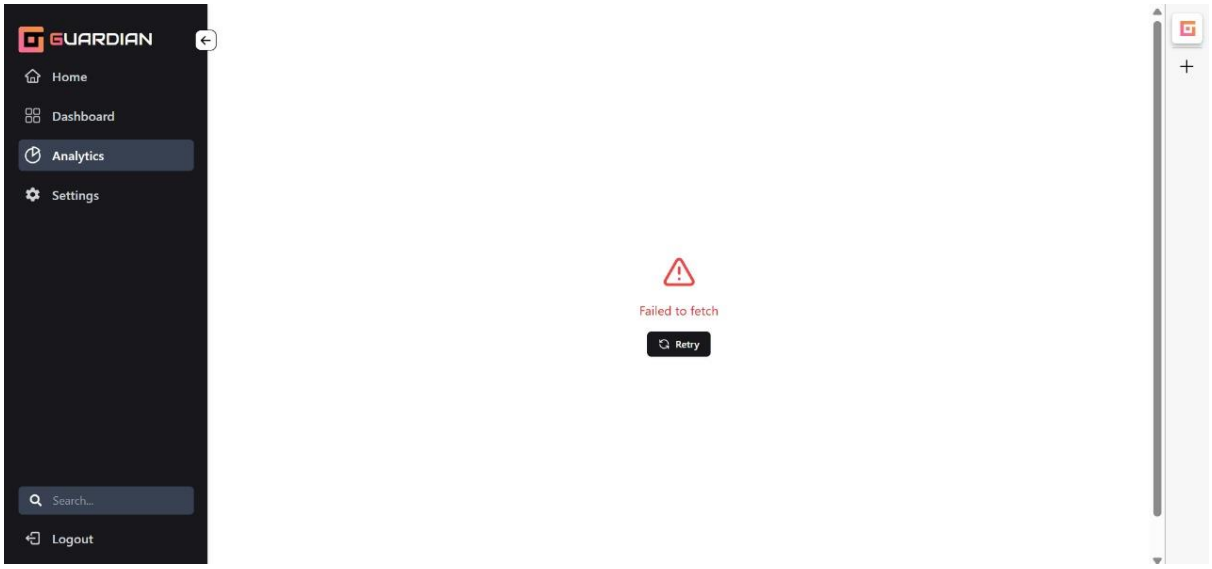
Picture 4 Analytic Page



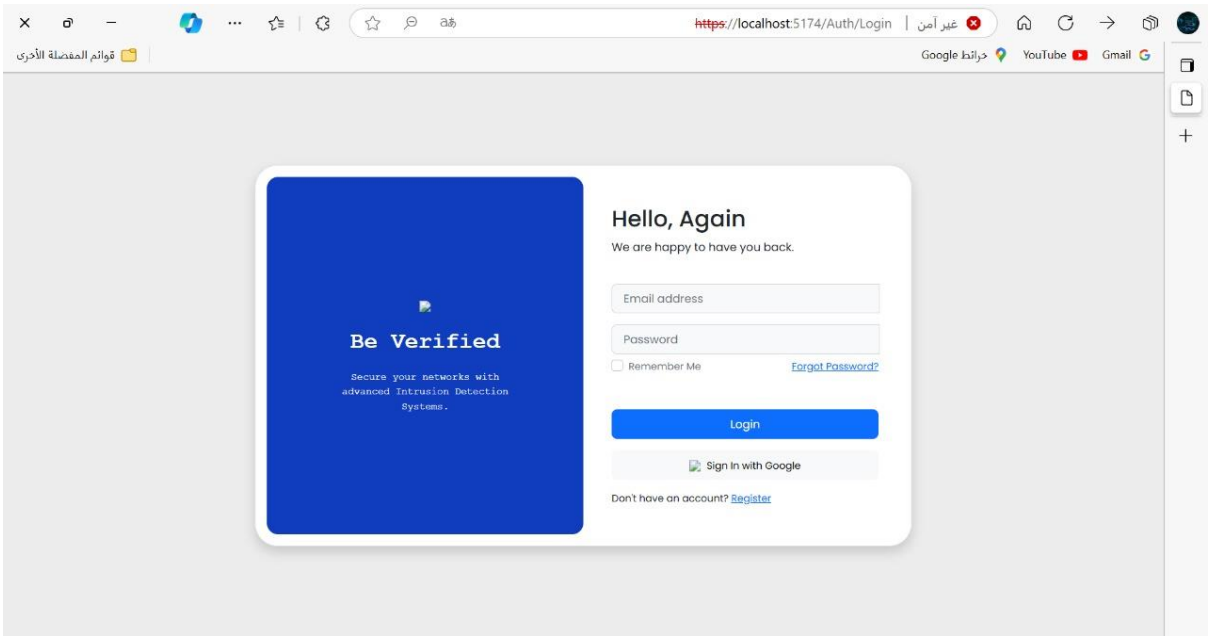
Picture 5 Analytic Page contd.



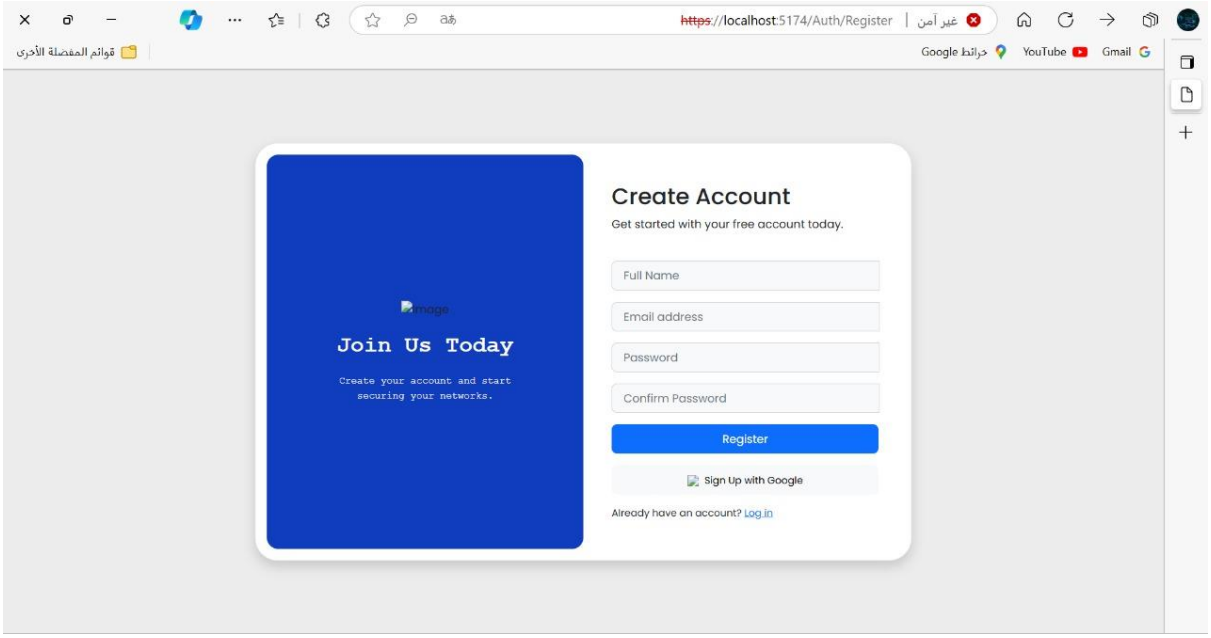
Picture 6 Dashboard - without log-in



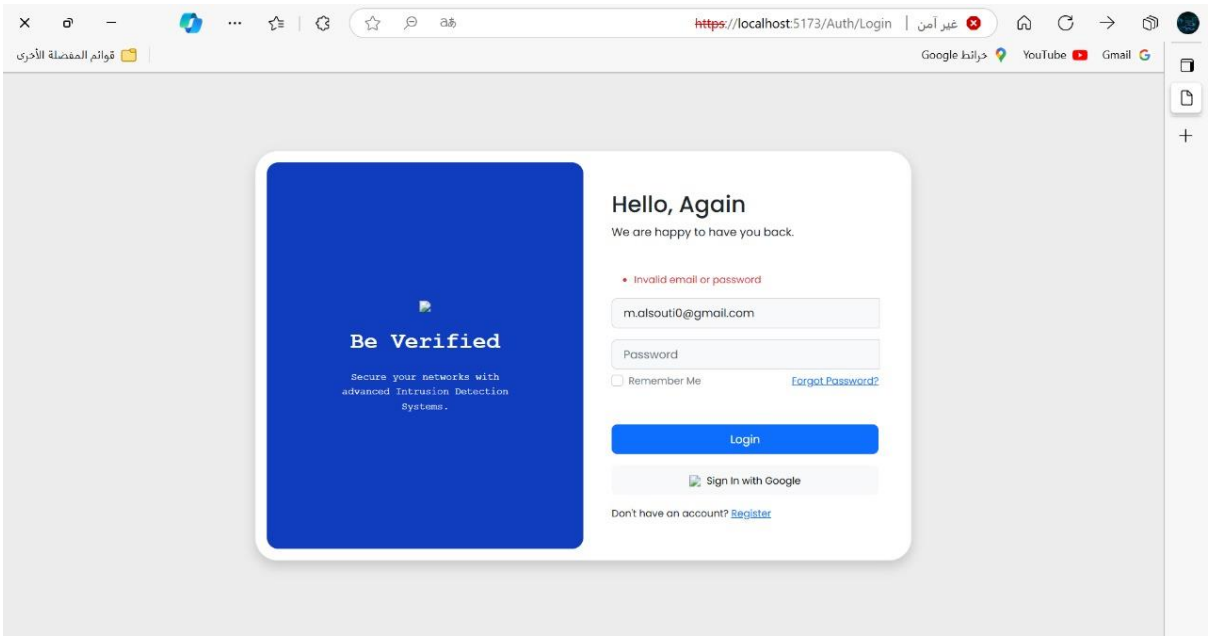
Picture 7 Analytics without log-in



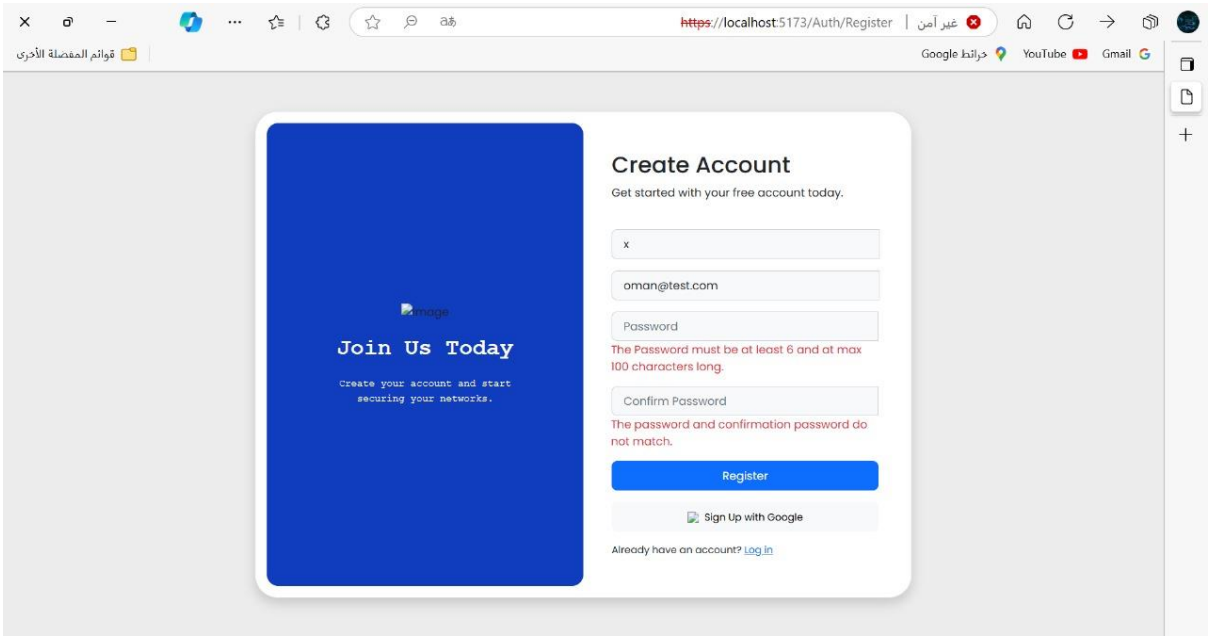
Picture 8 Log-in Page



Picture 9 Sign-up Page



Picture 10 log-in validation



Picture 11 sign-up validation

Implementation

Due to having many classes, there’s no point of copy-pasting the codes here. They will be submitted separately.

Here’s the explanation of the main classes we have for the MAIN functionalities.

Backend (ASP.NET)

The ASP.NET backend is designed to offer a RESTful API with several features for reporting, analytics, intrusion detection, and authentication. Here is a thorough explanation of each of its parts:

Authentication

Operations related to user login, registration, and sign-out are managed by the AuthController. By using hashed passwords to ensure security, the Login and Register endpoints authenticate users using their email address and password. The GetStatus endpoint gives real-time information on the user's authentication status, and if successful, the user is provided a session that is kept up to date via cookies. After successfully logging in, users may access the protected routes in the frontend thanks to the cookie-based authentication mechanism that uses ASP.NET Identity.

Analytics

The frontend may retrieve network statistics, traffic information, protocol distribution, and security packets thanks to the endpoints that the AnalyticsController offers. To receive data asynchronously, these API endpoints communicate with an IAnalyticsService. Additionally, they manage exceptions, report problems, and, in the event of a failure, provide an understandable error message. The GenerateReport endpoint provides customers with a downloadable file location after producing reports based on a specified period range.

Intrusion Detection

Network packets, intrusion patterns, and attack kinds can all be tracked with the IntrusionDetectionController. It facilitates packet analysis, including tracking packet details and trends, through interactions with an IPacketService. It enables users to disregard particular packets that are thought to be less important. The goal of this strategy is to identify network threats and make sure that possible intrusions are recorded and handled efficiently.

Data Models

The data supplied and received through the API is structured using a number of important data models, including PacketEntry, NetworkStats, TrafficData, SecurityPacket, and others. These models guarantee that the data is simply serialisable into JSON and is structured uniformly.

Frontend (REACT)

The interface through which users communicate with the backend is the React frontend. It is set up to communicate with the backend APIs while producing a clear and responsive user interface. This is a detailed explanation of the frontend's structure:

Components Structure:

For ease of maintenance, the parts are arranged into several directories:

- **Dashboard**: Consists of elements associated with the user dashboard, which shows security alerts, traffic data, and analytics.
- **Base**: Reusable elements including headers, footers, and layout structures are stored in this subdirectory.
- **Hooks**: To abstract logic like retrieving data from the backend and managing authentication, custom React hooks are defined here.
- **Services**: This directory includes services (such analytics and authentication) for dealing with the backend APIs.
- **UI**: The UI-related components, including buttons, cards, and form elements, are kept in this category.

React Router & Protected Routes

Only authenticated users can access some routes, such as the dashboard or settings, thanks to the ProtectedRoute component. This is accomplished by using a hook or the context to verify the user's authentication status. The user is taken to the login page if they are not authenticated.

Authentication

- The authentication context is defined by the AuthContext.tsx file, which enables the application to control the user's authentication status globally. To manage user authentication, it offers features like login, logout, and getCurrentUser.

- This context is used by components like Login.tsx and Register.tsx to carry out login and registration operations. Users are taken to the dashboard or the desired page after successfully logging in.

Dashboard & Analytics

Network statistics and important data visualisations are shown in the Dashboard component. Through a service file (such as Analytics.js), it interacts with the backend APIs to retrieve packet analysis, intrusion statistics, and network data. For ease of comprehension, the dashboard presents this data in an approachable way, most commonly with the use of tables or charts.
As an example, the Analytics shows:

- **Network statistics**, such as bandwidth usage and total packets.
- **Traffic information**, such as packets per time slot and risks identified
- **Trends in intrusions** (the kinds of intrusions found, how serious they are, and whether they are blocked)

Real-time data retrieved from the backend is the basis for the dynamic creation of these visualisations.

State Management

The data flow inside the application is managed using React's integrated state management, which makes use of the useState, useEffect, and context API. Services (such as those in services/Analytics.js or services/Auth.js) encapsulate API calls to the backend and centrally manage the replies for more intricate state interactions.

Integration with Backend

Through a sequence of API queries, the frontend communicates with the backend:

- The AuthController receives data from the frontend when a user registers or logs in, verifies the credentials, and then delivers an authentication status or token.
- The analytics and intrusion detection functions are available to the user after authentication. Requests for network statistics, traffic information, and intrusion trends are sent from the frontend to the AnalyticsController and IntrusionDetectionController.

Security Considerations

Security is a priority in the design of both the frontend and backend:

- Secure cookies are used on the backend for authentication, guaranteeing the safe storage of session data.
- To prevent sensitive information from being revealed, user passwords are kept on the server in a hashed format.
- React Router and ProtectedRoute on the frontend make sure that unauthorised users can't access the dashboard or other protected portions .

Details of the code

AnalyticsService.cs

```
using guardian_web_application.Server.Models;

namespace guardian_web_application.Server.Services
{
   public class AnalyticsService : IAnalyticsService
   {
      private readonly IPacketService _packetService;
      private readonly ILogger<AnalyticsService> _logger;

      public AnalyticsService(
         IPacketService packetService,
         ILogger<AnalyticsService> logger)
      {
         _packetService = packetService;
         _logger = logger;
      }

      public async Task<NetworkStats> GetNetworkStatsAsync()
      {
         var packets = await _packetService.GetPacketsAsync();
         var stats = new NetworkStats
         {
            TotalPackets = packets.Count(),
            HighRiskPackets = packets.Count(p => p.ThreatLevel == "high" || p.ThreatLevel
== "critical"),
            ActiveSessions = packets.GroupBy(p => p.Src).Count(),
            BandwidthUsage = CalculateBandwidthUsage(packets),
            UniqueIPs = GetUniqueIPs(packets),
            ThreatsBlocked = packets.Count(p => p.Attack != "None")
         };

         // Calculate changes (you would typically compare with historical data)
         stats.Changes = CalculateChanges(stats);
         return stats;
      }

      public async Task<IEnumerable<TrafficData>> GetTrafficDataAsync()
      {
```

```
        var packets = await _packetService.GetPacketsAsync();
        return packets
            .GroupBy(p => p.Timestamp.ToString("HH:00"))
            .Select(g => new TrafficData
            {
                TimeSlot = g.Key,
                Packets = g.Count(),
                Bandwidth = CalculateGroupBandwidth(g),
                Threats = g.Count(p => p.Attack != "None")
            })
            .OrderBy(t => t.TimeSlot);
    }

    public async Task<IEnumerable<ProtocolDistribution>>
GetProtocolDistributionAsync()
    {
        var packets = await _packetService.GetPacketsAsync();
        return packets
            .GroupBy(p => p.Service)
            .Select(g => new ProtocolDistribution
            {
                Name = g.Key,
                Value = g.Count()
            });
    }

    public async Task<IEnumerable<SecurityPacket>> GetSecurityPacketsAsync()
    {
        var packets = await _packetService.GetPacketsAsync();
        return packets
            .Where(p => p.Attack != "None")
            .Select(p => new SecurityPacket
            {
                Id = p.Id,
                IpSrc = p.Src,
                IpDst = p.Dst,
                Info = p.Info,
                Type = p.Attack,
                Status = "Unresolved",
                Severity = p.ThreatLevel,
                Timestamp = p.Timestamp
            });
    }

    public async Task<SecurityPacket> GetPacketDetailsAsync(int id)
    {
        var packets = await _packetService.GetPacketsAsync();
        var packet = packets.FirstOrDefault(p => p.Id == id);
        if (packet == null) return null;
```

```
    return new SecurityPacket
    {
        Id = packet.Id,
        IpSrc = packet.Src,
        IpDst = packet.Dst,
        Info = packet.Info,
        Type = packet.Attack,
        Status = "Unresolved",
        Severity = packet.ThreatLevel,
        Timestamp = packet.Timestamp
    };
}

public async Task<bool> ResolvePacketAsync(int id)
{
    return await _packetService.IgnorePacketAsync(id);
}

private double CalculateBandwidthUsage(IEnumerable<PacketEntry> packets)
{
    // Simplified calculation - you would typically have actual bandwidth data
    return packets.Count() * 0.001; // Assuming average packet size of 1KB
}

private int GetUniqueIPs(IEnumerable<PacketEntry> packets)
{
    return packets.Select(p => p.Src)
                  .Union(packets.Select(p => p.Dst))
                  .Distinct()
                  .Count();
}

private double CalculateGroupBandwidth(IEnumerable<PacketEntry> packets)
{
    return packets.Count() * 0.001;
}

private Dictionary<string, double> CalculateChanges(NetworkStats currentStats)
{
    // In a real application, you would compare with historical data
    return new Dictionary<string, double>
    {
        ["TotalPackets"] = 12.5,
        ["HighRiskPackets"] = -5.2,
        ["ActiveSessions"] = 8.7,
        ["BandwidthUsage"] = 15.3,
        ["UniqueIPs"] = 10.1,
        ["ThreatsBlocked"] = -8.9
    };
}
```

```
    }
}
```

1-Dependencies and Constructor:

- AnalyticsService depends on IPacketService to retrieve packets and on ILogger for logging.
- Initializing these dependencies constructor.

2-Network Statistics (GetNetworkStatsAsync):

- Generate network statistics from the packet data.
- Key metrics include:
    - Total packets.
    - High-risk packets.
    - Active sessions (unique sources).
    - Estimated bandwidth usage.
    - Unique IP addresses.
    - Blocked threats.
- Returns a NetworkStats object contains this information.

3-Traffic Data (GetTrafficDataAsync):

- This function classifies packets by hourly time slots, counts packets, estimates bandwidth, and tracking threats in each time slot.
- Returns an ordered list of TrafficData objects.

4-Protocol Distribution (GetProtocolDistributionAsync):

- Groups packets by service type or the protocol.
- Returns a list of ProtocolDistribution objects illustrating the distribution of different protocols.

5-Security Packets (GetSecurityPacketsAsync):

- Filtering the packets marked as security incidents.
- Returns a list of SecurityPacket objects containing details like IPs, attack type, and threat level.

6-Packet Details (GetPacketDetailsAsync):

- Returns detailed information for a packet by its ID.
- Returns a SecurityPacket object or null if not found.

7-Resolve Packet (ResolvePacketAsync):

- Marking packet as “resolved” by ignoring it using the IPacketService.

8-Helper Methods:

- CalculateBandwidthUsage: Calculates estimated bandwidth depending on packet count.
- GetUniqueIPs: Get number of unique IP addresses by source and destination.
- CalculateGroupBandwidth: Estimates bandwidth for a group of packets.
- CalculateChanges: Provides changes in statistics for analysis by comparing current stats with hypothetical historical data.

PacketService.cs

```
using CsvHelper;

using guardian_web_application.Server.Models;

using Microsoft.Extensions.Configuration;
```

```
using System.Formats.Asn1;
using System.Globalization;

namespace guardian_web_application.Server.Services
{
    public class PacketService : IPacketService
    {
        private readonly string _csvPath;
        private readonly ILogger<PacketService> _logger;

        public PacketService(IConfiguration configuration, ILogger<PacketService> logger)
        {
            _logger = logger;
            string baseDir = AppDomain.CurrentDomain.BaseDirectory;
            string dataDir = Path.Combine(baseDir, "Data");
            _csvPath = Path.Combine(dataDir, "packets.csv");
            Directory.CreateDirectory(dataDir);

            if (!File.Exists(_csvPath))
            {
                _logger.LogInformation("Creating new sample data file at: " + _csvPath);
                CreateSampleData();
            }
        }
```

```
    private void CreateSampleData()
    {
        try
        {
            var random = new Random();
            var startDate = new DateTime(2023, 1, 1);
            var attackTypes = new[] { "None", "DDoS", "Brute Force", "SQL Injection",
"XSS", "Remote Code Execution",
                                      "Man in Middle", "Malware", "Path Traversal", "Data Exfiltration",
"Reconnaissance", "CSRF" };
            var services = new[] { "HTTP", "HTTPS", "FTP", "SSH", "SMTP", "MySQL",
"RDP", "DNS" };
            var ports = new Dictionary<string, int> {
                { "HTTP", 80 }, { "HTTPS", 443 }, { "FTP", 21 }, { "SSH", 22 },
                { "SMTP", 25 }, { "MySQL", 3306 }, { "RDP", 3389 }, { "DNS", 53 }
            };
            var threatLevels = new[] { "low", "medium", "high", "critical" };

            var sampleData = new List<PacketEntry>();
            var id = 1;

            // Create entries for each month
            for (int month = 1; month <= 12; month++)
            {
                // Generate more entries for peak activity months
                int entriesPerMonth = month >= 6 && month <= 8 ? 20 : 15;
```

```
            for (int entry = 1; entry <= entriesPerMonth; entry++)
            {
                var day = random.Next(1, DateTime.DaysInMonth(2023, month) + 1);
                var timestamp = new DateTime(2023, month, day);
                var attack = attackTypes[random.Next(attackTypes.Length)];
                var service = services[random.Next(services.Length)];
                var threatLevel = attack == "None" ? "low" : threatLevels[random.Next(1,
threatLevels.Length)];

                sampleData.Add(new PacketEntry
                {
                    Id = id++,
                    Src = $"192.168.1.{random.Next(1, 255)}",
                    Dst = $"192.168.1.{random.Next(1, 255)}",
                    Port = ports[service],
                    Service = service,
                    Info = GenerateInfo(attack),
                    Attack = attack,
                    ThreatLevel = threatLevel,
                    Timestamp =
timestamp.AddHours(random.Next(24)).AddMinutes(random.Next(60))
                });
            }
        }

        // Sort by timestamp
```

```
            sampleData = sampleData.OrderBy(x => x.Timestamp).ToList();


            // Write to CSV
            using (var writer = new StreamWriter(_csvPath, false,
System.Text.Encoding.UTF8))
            using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))
            {
                csv.WriteRecords(sampleData);
            }


            _logger.LogInformation($"Successfully created sample data with
{sampleData.Count} entries");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error creating sample data file");
            throw;
        }
    }


    private string GenerateInfo(string attack)
    {
        return attack switch
        {
            "None" => "Normal access",
            "DDoS" => "High traffic volume detected",
```

```
        "Brute Force" => "Multiple login attempts",
        "SQL Injection" => "Suspicious SQL query pattern",
        "XSS" => "Cross-site scripting attempt",
        "Remote Code Execution" => "Suspicious command execution",
        "Man in Middle" => "SSL/TLS anomaly detected",
        "Malware" => "Malicious payload detected",
        "Path Traversal" => "Directory traversal attempt",
        "Data Exfiltration" => "Unusual data transfer pattern",
        "Reconnaissance" => "System scanning detected",
        "CSRF" => "Cross-site request forgery attempt",
        _ => "Unknown activity"
    };
}

public async Task<IEnumerable<PacketEntry>> GetPacketsAsync()
{
    try
    {
        var packets = new List<PacketEntry>();
        using (var reader = new StreamReader(_csvPath))
        using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
        {
            packets = csv.GetRecords<PacketEntry>().ToList();
        }
        return packets;
```

```
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error reading packet data from CSV");
            throw;
        }
    }

    public async Task<IDictionary<string, int>> GetPacketStatsAsync()
    {
        var packets = await GetPacketsAsync();
        return new Dictionary<string, int>
        {
            ["TotalPackets"] = packets.Count(),
            ["Attacks"] = packets.Count(p => p.Attack != "None"),
            ["Intrusions"] = packets.Count(p => p.ThreatLevel == "high" || p.ThreatLevel ==
"critical"),
            ["ActiveThreats"] = packets.Count(p => p.ThreatLevel != "low")
        };
    }

    public async Task<IEnumerable<IntrusionTrend>> GetIntrusionTrendsAsync()
    {
        var packets = await GetPacketsAsync();
        return packets
            .GroupBy(p => p.Timestamp.ToString("MMM"))
```

```
            .Select(g => new IntrusionTrend
            {
                Name = g.Key,
                Detected = g.Count(),
                Blocked = g.Count(p => p.ThreatLevel != "critical"),
                Average = g.Average(p => p.ThreatLevel == "critical" ? 1 : 0) * 100
            })
            .OrderBy(t => DateTime.ParseExact(t.Name, "MMM",
CultureInfo.InvariantCulture))
            .ToList();
        }

        public async Task<IEnumerable<IntrusionType>> GetIntrusionTypesAsync()
        {
            var packets = await GetPacketsAsync();
            return packets
                .Where(p => p.Attack != "None")
                .GroupBy(p => p.Attack)
                .Select(g => new IntrusionType
                {
                    Name = g.Key,
                    Count = g.Count(),
                    Severity = g.Max(p => p.ThreatLevel),
                    Risk = CalculateRisk(g.ToList())
                })
                .OrderByDescending(t => t.Count)
```

```
        .ToList();
}

public async Task<bool> IgnorePacketAsync(int id)
{
    try
    {
        var packets = (await GetPacketsAsync()).ToList();
        var updatedPackets = packets.Where(p => p.Id != id);

        using (var writer = new StreamWriter(_csvPath))
        using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))
        {
            csv.WriteRecords(updatedPackets);
        }

        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error ignoring packet");
        return false;
    }
}
```

```
    private int CalculateRisk(List<PacketEntry> packets)
    {
        var threatLevelWeights = new Dictionary<string, int>
        {
            ["low"] = 25,
            ["medium"] = 50,
            ["high"] = 75,
            ["critical"] = 100
        };

        return (int)packets.Average(p => threatLevelWeights[p.ThreatLevel.ToLower()]);
    }

    public string GetServiceFromPort(int port)
    {
        if (Enum.IsDefined(typeof(PortService), port))
        {
            return ((PortService)port).ToString();
        }
        return "Unknown";
    }

  }

}
```

1. **Dependencies and Initialization**:
    - PacketService depends on IConfiguration for the configuration settings and ILogger for logging.
    - The constructor sets the path of the CSV file to store packet data and creates the data directory if it doesn't exist.
    - If the CSV file is missing, it calls CreateSampleData to initialize sample packet data.
2. **Creating Sample Data (CreateSampleData)**:
    - Generates some random packet entries for each month in 2023.
    - Each packet contain fields like source/destination IPs, port, service, threat level if it is a threat, timestamp, and attack type in case it is an attack.
    - Writes back the generated data to a CSV file to use it later by other service methods.
3. **Helper Method: GenerateInfo**:
    - Provides short descriptions for different attack types, such as "High traffic volume detected" for DDoS or "Multiple login attempts" for Brute Force.
4. **Retrieving Packet Data (GetPacketsAsync)**:
    - Reads the packet data from the CSV file and returns it as a list of PacketEntry objects.
5. **Statistics Retrieval (GetPacketStatsAsync)**:
    - Use the data to create basic statistics:
        - Total packets.
        - Total attacks.
        - High/critical threat level intrusions.
        - Non-low threat packets (active threats).
6. **Intrusion Trends (GetIntrusionTrendsAsync)**:
    - Clusters packets by month, calculating:
        - Total detected intrusions.
        - Blocked intrusions.
        - Percentage of critical threats.
7. **Intrusion Types (GetIntrusionTypesAsync)**:
    - Clusters packets by attack type and calculates:
        - Count for each type.
        - Maximum severity level for each type.
        - Risk score using the CalculateRisk helper method.
8. **Ignoring a Packet (IgnorePacketAsync)**:
    - Removing a packet by it is ID from the CSV file by filtering it out and then overwriting the CSV file with the packets after it.
9. **Helper Method: CalculateRisk**:
    - Calculates an average risk score for a group of packets depending on threat levels, assigning weights to levels from "low" to "critical".
10. **Service Lookup by Port (GetServiceFromPort)**:

- Mapping between port number to a service name based on a predefined port-service mappings, returning "Unknown" if no match is found.

The PacketService class define methods to manage and analyze packet data, supporting some functionalities of fetching data, calculating trends, and filtering out packets. It relies on CSV data storage and uses CsvHelper to handle reading/writing CSV files.

register.cshtml.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.RegularExpressions;

namespace guardian_web_application.Server.Validation
{
    public class StrongPasswordAttribute : ValidationAttribute
    {
        public int MinimumLength { get; set; } = 8;
        public bool RequireDigit { get; set; } = true;
        public bool RequireLowercase { get; set; } = true;
        public bool RequireUppercase { get; set; } = true;
        public bool RequireSpecialCharacter { get; set; } = true;
        public bool DisallowCommonPasswords { get; set; } = true;

        protected override ValidationResult? IsValid(object? value, ValidationContext
validationContext)
        {
            if (value == null || string.IsNullOrWhiteSpace(value.ToString()))
            {
                return new ValidationResult("Password cannot be empty.");
            }

            var password = value.ToString();
            var errors = new List<string>();
```

```
        if (password.Length < MinimumLength)
            errors.Add($"Password must be at least {MinimumLength} characters long.");

        if (RequireDigit && !password.Any(char.IsDigit))
            errors.Add("Password must contain at least one digit.");

        if (RequireLowercase && !password.Any(char.IsLower))
            errors.Add("Password must contain at least one lowercase letter.");

        if (RequireUppercase && !password.Any(char.IsUpper))
            errors.Add("Password must contain at least one uppercase letter.");

        if (RequireSpecialCharacter && !Regex.IsMatch(password,
@"[!@#$%^&*(),.?\"":{}|<>]"))
            errors.Add("Password must contain at least one special character.");

        if (DisallowCommonPasswords && IsCommonPassword(password))
            errors.Add("This password is too common. Please choose a more unique
password.");

        // Check for sequential characters
        if (HasSequentialCharacters(password))
            errors.Add("Password cannot contain sequential characters (e.g., '123', 'abc').");

        // Check for repeating characters
```

```
    if (HasRepeatingCharacters(password))
        errors.Add("Password cannot contain repeating characters (e.g., '111', 'aaa').");

    return errors.Count == 0
        ? ValidationResult.Success
        : new ValidationResult(string.Join(" ", errors));
}

private bool IsCommonPassword(string password)
{
    // Add a list of common passwords to check against
    var commonPasswords = new HashSet<string>
    {
        "password123", "12345678", "qwerty123", "admin123",
        "letmein123", "welcome123", "monkey123", "football123"
        // Add more common passwords as needed
    };

    return commonPasswords.Contains(password.ToLower());
}

private bool HasSequentialCharacters(string password)
{
    const string sequences = "abcdefghijklmnopqrstuvwxyz01234567890";
    const int sequenceLength = 3;
```

```
        for (int i = 0; i <= sequences.Length - sequenceLength; i++)
        {
            string forward = sequences.Substring(i, sequenceLength);
            string backward = new string(forward.Reverse().ToArray());

            if (password.ToLower().Contains(forward) ||
password.ToLower().Contains(backward))
                return true;
        }

        return false;
    }

    private bool HasRepeatingCharacters(string password)
    {
        const int maxRepeats = 3;
        for (int i = 0; i <= password.Length - maxRepeats; i++)
        {
            if (password.Skip(i).Take(maxRepeats).Distinct().Count() == 1)
                return true;
        }

        return false;
    }
}
```

}

1. **Properties**:
    - The attribute includes define password requirements:
        - MinimumLength: Minimum length of the password (default: 8).
        - RequireDigit: Ensures at least one numeric digit.
        - RequireLowercase: Ensures at least one lowercase letter.
        - RequireUppercase: Ensures at least one uppercase letter.
        - RequireSpecialCharacter: Ensures at least one special character.
        - DisallowCommonPasswords: Disallows commonly used passwords for increased security.
2. **Validation (IsValid Method)**:
    - Validates a password based on the attributes defined above:
        - Checks if the password meets the requirements.
        - Checks if the password is in commonly used passwords (based on a predefined list).
        - Checks for sequential characters (e.g., "123", "abc").
        - Checks for repeating characters (e.g., "111", "aaa").
    - Returns a ValidationResult.Success if the password meets all requirements; If not, it returns a message summarizing the issues.
3. **Helper Methods**:
    - IsCommonPassword: Checks if the password is in a set of common passwords (e.g., "password123").
    - HasSequentialCharacters: Scans for any three-character sequence in row, like "abc" or "123", both forwards and backwards.
    - HasRepeatingCharacters: Detects if any character repeats more than three times consecutively.

This aim of this class is to enhance password security by enforcing complex password rules.

AuthController.cs

```
using Microsoft.AspNetCore.Mvc;

using System.Security.Claims;

using Microsoft.AspNetCore.Authentication;

using Microsoft.AspNetCore.Authentication.Cookies;

using guardian_web_application.Server.Services;

using System.ComponentModel.DataAnnotations;

using Microsoft.AspNetCore.Authorization;



namespace guardian_web_application.Server.Controllers
```

```
{
    [Authorize]
    [ApiController]
    [Route("api/[controller]")]
    public class AuthController : ControllerBase
    {
        private readonly IUserService _userService;
        private readonly IConfiguration _configuration;
        private readonly ILogger<AuthController> _logger;

        public class LoginRequestModel
        {
            public string Email { get; set; } = string.Empty;
            public string Password { get; set; } = string.Empty;
            public bool RememberMe { get; set; }
        }

        public class RegisterRequestModel
        {
            [Required]
            [EmailAddress]
            public string Email { get; set; } = string.Empty;

            [Required]
            [StringLength(100, MinimumLength = 6)]
```

```
    public string Password { get; set; } = string.Empty;

    [Required]
    [Compare("Password")]
    public string ConfirmPassword { get; set; } = string.Empty;

    [Required]
    public string FullName { get; set; } = string.Empty;
}

public class AuthResponseModel
{
    public bool Success { get; set; }
    public string Message { get; set; } = string.Empty;
    public string? RedirectUrl { get; set; }
}

public AuthController(
    IUserService userService,
    IConfiguration configuration,
    ILogger<AuthController> logger)
{
    _userService = userService;
    _configuration = configuration;
    _logger = logger;
```

```
    }

    [HttpGet("status")]
    public IActionResult GetStatus()
    {
        var isAuthenticated = User.Identity?.IsAuthenticated ?? false;
        return Ok(new
        {
            isAuthenticated,
            user = isAuthenticated ? new
            {
                email = User.FindFirst(ClaimTypes.Email)?.Value,
                name = User.FindFirst(ClaimTypes.Name)?.Value,
                role = User.FindFirst(ClaimTypes.Role)?.Value
            } : null
        });
    }

    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody] LoginRequestModel request)
    {
        try
        {
            var (success, message, user) = await
_userService.ValidateUserAsync(request.Email, request.Password);
```

```
        if (!success || user == null)
        {
            return BadRequest(new AuthResponseModel
            {
                Success = false,
                Message = message
            });
        }

        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name, user.FullName),
            new Claim(ClaimTypes.Email, user.Email),
            new Claim(ClaimTypes.NameIdentifier, user.Id.ToString())
        };

        var claimsIdentity = new ClaimsIdentity(claims,
CookieAuthenticationDefaults.AuthenticationScheme);
        var authProperties = new AuthenticationProperties
        {
            IsPersistent = request.RememberMe,
            ExpiresUtc = DateTimeOffset.UtcNow.AddDays(7)
        };

        await HttpContext.SignInAsync(
            CookieAuthenticationDefaults.AuthenticationScheme,
```

```
                new ClaimsPrincipal(claimsIdentity),
                authProperties);

            return Ok(new AuthResponseModel
            {
                Success = true,
                Message = "Login successful",
                RedirectUrl = $"{_configuration["Spa:ClientUrl"]}/app"
            });
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Login failed");
            return StatusCode(500, new AuthResponseModel
            {
                Success = false,
                Message = "An error occurred during login"
            });
        }
    }

    [HttpPost("register")]
    public async Task<IActionResult> Register([FromBody] RegisterRequestModel
request)
    {
        try
```

```
{
    if (request.Password != request.ConfirmPassword)
    {
        return BadRequest(new AuthResponseModel
        {
            Success = false,
            Message = "Passwords do not match"
        });
    }

    var (success, message, user) = await _userService.RegisterUserAsync(
        request.Email,
        request.Password,
        request.FullName);

    if (!success || user == null)
    {
        return BadRequest(new AuthResponseModel
        {
            Success = false,
            Message = message
        });
    }

    // After successful registration, redirect to login instead of auto-login
```

```
            return Ok(new AuthResponseModel
            {
                Success = true,
                Message = "Registration successful. Please login.",
                RedirectUrl = "https://localhost:7024/Auth/Login"
            });
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Registration failed");
            return StatusCode(500, new AuthResponseModel
            {
                Success = false,
                Message = "An error occurred during registration"
            });
        }
    }

    [HttpPost("sign-out")]
    public  async Task<SignOutResult> SignOut()
    {
        try
        {
            await
HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
```

```
            // Clear authentication cookies
            foreach (var cookie in Request.Cookies.Keys)
            {
                Response.Cookies.Delete(cookie);
            }

            return SignOut(new AuthenticationProperties
            {
                RedirectUri = "/Auth/Login"
            }, CookieAuthenticationDefaults.AuthenticationScheme);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Logout failed");
            throw;
        }
    }
  }
}
```

- **Route:** api/auth
- **Authorization:** The controller is protected with [Authorize], that means all actions require the user to be authenticated.

Key Actions:

1. **GetStatus** (GET /status):
    - Checks the current authentication status of the user. If authenticated, it returns the user's details.
2. **Login** (POST /login):
    - Accepts user credentials (email and password), validates them through the IUserService, and if they are valid, signs the user in using cookie authentication.
    - Returns a success message then redirect URL to the client-side application.

    - If authentication fails, it shows an error message.
3. **Register** (POST /register):
    - Accepts user registration details (email, password, full name). It validates if the password and password confirmation matches and registers the user via IUserService.
    - Returns a success message then redirect URL to the login page if registration succeeds.
    - If fails, it shows error details.
4. **SignOut** (POST /sign-out):
    - Logs out the user by signing out of the current authentication scheme and remove authentication cookies.
    - Redirects the user to the login page after that.

Models:

- **LoginRequestModel**: Captures login details (email, password, remember me option).
- **RegisterRequestModel**: Captures registration details (email, password, full name, and confirm password).
- **AuthResponseModel**: Used for showing the success status, message, and if there is any redirect URL.

Dependencies:

- **IUserService**: A service for user registration and validation logic.
- **ILogger<AuthController>**: Used for logging and keep track the errors and events.
- **IConfiguration**: Allows access to configuration options, such as client URLs for redirects.

This controller ensures that the web application's user authentication and authorization processes are handled securely and efficiently.

Team member	Percentage	Work complete
Mojahid	40%	• Creating the project layout • linking back end and front end
Amr	30%	• writing the report and ERD • Configure some front end pages
Osama	30%	• Define the classes and validations for each • Define the Controllers