Comprehensive BSN Dashboard Implementation Guide

This guide provides detailed, step-by-step instructions for creating a complete Bulgarian Society Netherlands (BSN) dashboard application in Unity with real data integration, proper authentication, and all the features shown in your prototype.

Table of Contents

---

1. Project Setup

1.1 Creating the Unity Project

1. **Download Unity Hub and Unity**
   - Install Unity Hub from the official website
   - Install Unity version 2021.3 LTS (or newer stable LTS version)

2. **Create a new Unity Project**

```
Project Name: BSN_Dashboard
Template: 2D
Location: [Choose your workspace folder]
```

3. **Configure project settings**
   - Open the project and go to Edit > Project Settings
   - Player Settings:
     - Set Company Name to "BSN"
     - Set Product Name to "BSN Dashboard"
     - Set Default Screen Orientation to Portrait
     - Enable "Auto Graphics API" for Android
   - Quality Settings:
     - Set Quality Level to "Medium" for mobile optimization

4. **Install required packages**
   - Open Window > Package Manager
   - Install the following packages:
     - TextMeshPro
     - Unity UI
     - Addressables (for asset management)
     - UniTask (from Git URL: https://github.com/Cysharp/UniTask.git?path=src/UniTask/Assets/Plugins/UniTask)
     - Newtonsoft Json (from Package Manager)

1.2 Project Structure Setup

1. **Create folder structure in Assets**

```
/Scripts
  /Data
  /UI
  /Networking
  /Auth
  /Utilities
  /Services
/Prefabs
  /UI
  /Popups
/Resources
  /Textures
  /Fonts
  /Audio
/Scenes
/StreamingAssets
/AddressableAssets
```

2. **Configure version control**
   - Create a .gitignore file (if using Git)
   - Initialize Git repository (optional but recommended)
   - Make initial commit
3. **Set up scene structure**
   - Create and save the following scenes:
     - `SplashScreen.unity`
     - `Login.unity`
     - `Dashboard.unity`

2. UI Framework Implementation

2.1 Setting Up Canvas and Layout System

1. **Create base Canvas in the Dashboard scene**

```
Hierarchy > Create > UI > Canvas
```

- Set Canvas Scaler to "Scale With Screen Size"
- Reference Resolution: 1080x1920
- Screen Match Mode: Match Width or Height
- Match: 0.5 (balance between width and height)

2. **Create a UI Manager script**

csharp

```csharp
// UIManager.cs
using UnityEngine;
using System.Collections.Generic;

public class UIManager : MonoBehaviour
{
    // Singleton pattern
    public static UIManager Instance { get; private set; }

    [SerializeField] private GameObject loadingScreenPrefab;
    [SerializeField] private Transform canvasTransform;

    private Dictionary<string, GameObject> activePanels = new Dictionary<string, GameObject
    private GameObject loadingScreen;

    private void Awake()
    {
        // Singleton setup
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }

        // Initialize loading screen
        loadingScreen = Instantiate(loadingScreenPrefab, canvasTransform);
        loadingScreen.SetActive(false);
    }
```

```
    public void ShowPanel(string panelName)
    {
        foreach (var panel in activePanels.Values)
        {
            panel.SetActive(false);
        }

        if (activePanels.TryGetValue(panelName, out GameObject panel))
        {
            panel.SetActive(true);
        }
    }

    public void RegisterPanel(string panelName, GameObject panel)
    {
        if (!activePanels.ContainsKey(panelName))
        {
            activePanels.Add(panelName, panel);
        }
    }

    public void ShowLoading(bool show)
    {
        loadingScreen.SetActive(show);
    }

    // Add additional UI utility methods as needed
}
```

3. **Create UI Prefabs**
   - Create prefabs for common UI elements:
     - Button
     - Panel
     - Input Field
     - Dropdown
     - Custom components (like metric displays)

2.2 Implementing Navigation System

1. **Create a Navigation Manager script**

csharp

```csharp
// NavigationManager.cs
using UnityEngine;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
using Cysharp.Threading.Tasks;

public class NavigationManager : MonoBehaviour
{
    public static NavigationManager Instance { get; private set; }

    [SerializeField] private GameObject[] mainPanels;
    [SerializeField] private GameObject menuPanel;

    private Stack<GameObject> navigationHistory = new Stack<GameObject>();
    private GameObject currentPanel;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }
    }

    public void NavigateTo(GameObject targetPanel)
    {
        // Hide current panel
        if (currentPanel != null)
        {
            currentPanel.SetActive(false);
            navigationHistory.Push(currentPanel);
        }

        // Show target panel
        targetPanel.SetActive(true);
        currentPanel = targetPanel;
    }
```

```
    public bool NavigateBack()
    {
        if (navigationHistory.Count > 0)
        {
            // Hide current panel
            currentPanel.SetActive(false);

            // Show previous panel
            currentPanel = navigationHistory.Pop();
            currentPanel.SetActive(true);

            return true;
        }

        return false;
    }

    public void ToggleMenu()
    {
        menuPanel.SetActive(!menuPanel.activeSelf);
    }

    public async UniTask LoadScene(string sceneName)
    {
        UIManager.Instance.ShowLoading(true);

        // Clear navigation history when switching scenes
        navigationHistory.Clear();

        await SceneManager.LoadSceneAsync(sceneName);

        UIManager.Instance.ShowLoading(false);
    }
}
```

2. **Create Menu Panel prefab**
   - Design a hamburger menu panel prefab with options:
     - Dashboard
     - Cities
     - Charts
     - Achievements
     - Settings
     - Logout

2.3 Implementing the Login Screen

1. **Design Login UI**
   - Create TextMeshPro input fields for username and password
   - Add login button
   - Add BSN logo
   - Add error message text (initially hidden)

2. **Create Login Controller script**

csharp

```csharp
// LoginController.cs
using UnityEngine;
using TMPro;
using UnityEngine.UI;
using Cysharp.Threading.Tasks;

public class LoginController : MonoBehaviour
{
    [SerializeField] private TMP_InputField usernameField;
    [SerializeField] private TMP_InputField passwordField;
    [SerializeField] private Button loginButton;
    [SerializeField] private TextMeshProUGUI errorText;

    private void Start()
    {
        errorText.gameObject.SetActive(false);
        loginButton.onClick.AddListener(AttemptLogin);

        // Check if we have stored credentials
        TryAutoLogin();
    }

    private void TryAutoLogin()
    {
        if (AuthManager.Instance.HasStoredCredentials())
        {
            // Auto-login without user input
            AuthManager.Instance.LoginWithStoredCredentials().Forget();
        }
    }

    private async void AttemptLogin()
    {
        // Disable button during login
        loginButton.interactable = false;
        errorText.gameObject.SetActive(false);
```

```
        string username = usernameField.text;
        string password = passwordField.text;

        if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password))
        {
            ShowError("Please enter username and password");
            loginButton.interactable = true;
            return;
        }

        // Attempt login
        bool success = await AuthManager.Instance.Login(username, password);

        if (success)
        {
            // Navigate to dashboard scene
            await NavigationManager.Instance.LoadScene("Dashboard");
        }
        else
        {
            ShowError("Invalid username or password");
            loginButton.interactable = true;
        }
    }

    private void ShowError(string message)
    {
        errorText.text = message;
        errorText.gameObject.SetActive(true);
    }
}
```

3. Data Architecture

3.1 Core Data Models

1. **Create base data models**

csharp

```csharp
// Models/SocialMediaMetrics.cs
using System;

[Serializable]
public class SocialMediaMetrics
{
    public int instagramFollowers;
    public int tiktokFollowers;
    public int tiktokLikes;

    // Constructor for testing
    public SocialMediaMetrics(int igFollowers = 0, int ttFollowers = 0, int ttLikes = 0)
    {
        instagramFollowers = igFollowers;
        tiktokFollowers = ttFollowers;
        tiktokLikes = ttLikes;
    }
}

// Models/EventMetrics.cs
using System;

[Serializable]
public class EventMetrics
{
    public int ticketsSold;
    public float averageAttendance;
    public int totalEvents;

    // Constructor for testing
    public EventMetrics(int tickets = 0, float avgAttendance = 0, int events = 0)
    {
        ticketsSold = tickets;
        averageAttendance = avgAttendance;
        totalEvents = events;
    }
}

// Models/Achievement.cs
using System;

[Serializable]
public class Achievement
```

```
{
    public string id;
    public string title;
    public string description;
    public int currentValue;
    public int targetValue;
    public bool isUnlocked;
    public DateTime unlockDate;

    public float Progress => (float)currentValue / targetValue;
}

// Models/CityData.cs
using System;
using System.Collections.Generic;

[Serializable]
public class CityData
{
    public string cityName;
    public SocialMediaMetrics socialMedia;
    public EventMetrics events;
    public List<Achievement> achievements;
    public DateTime lastUpdated;

    // Additional properties for display
    public string flagColor; // Hex color for city-specific styling

    // Constructor with default values
    public CityData()
    {
        socialMedia = new SocialMediaMetrics();
        events = new EventMetrics();
        achievements = new List<Achievement>();
        lastUpdated = DateTime.Now;
    }
}

// Models/HistoricalDataPoint.cs
using System;

[Serializable]
public class HistoricalDataPoint
{
    public DateTime date;
    public string metricName;
    public double value;
```

```
    public string cityName;

    public HistoricalDataPoint(DateTime date, string metricName, double value, string cityN
    {
        this.date = date;
        this.metricName = metricName;
        this.value = value;
        this.cityName = cityName;
    }
}

// Models/UserProfile.cs
using System;

[Serializable]
public class UserProfile
{
    public string username;
    public string displayName;
    public string cityAssignment; // "all" for main board, or specific city name
    public bool isMainBoard;
    public string[] permissions;

    public bool CanAccessCity(string cityName)
    {
        return isMainBoard || cityAssignment.Equals(cityName, StringComparison.OrdinalIgnor
    }
}
```

2. **Create Data Service interfaces**

csharp

```
// Services/IDataService.cs
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

public interface IDataService
{
    Task<CityData> GetCityDataAsync(string cityName);
    Task<List<string>> GetAllCityNamesAsync();
    Task<List<HistoricalDataPoint>> GetHistoricalDataAsync(string cityName, string metricNa
    Task<bool> UpdateAchievementProgressAsync(string cityName, string achievementId, int ne
}
```

3.2 Data Management

1. **Create Data Manager class**

csharp

```csharp
// DataManager.cs
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Cysharp.Threading.Tasks;

public class DataManager : MonoBehaviour
{
    public static DataManager Instance { get; private set; }

    [SerializeField] private float autoRefreshInterval = 300f; // 5 minutes

    private IDataService dataService;
    private ICacheService cacheService;

    // Current data
    private Dictionary<string, CityData> cityDataCache = new Dictionary<string, CityData>()
    private Dictionary<string, List<HistoricalDataPoint>> historicalDataCache = new Diction

    // Event for notifying subscribers of data updates
    public event Action<string> OnCityDataUpdated;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }
    }

    public void Initialize(IDataService dataService, ICacheService cacheService)
    {
        this.dataService = dataService;
        this.cacheService = cacheService;

        // Start auto-refresh coroutine
```

```
    StartAutoRefresh().Forget();
}

private async UniTaskVoid StartAutoRefresh()
{
    while (true)
    {
        // Wait for the specified interval
        await UniTask.Delay(TimeSpan.FromSeconds(autoRefreshInterval));

        // Only refresh if user is logged in
        if (AuthManager.Instance.IsLoggedIn)
        {
            await RefreshAllData();
        }
    }
}

public async UniTask<CityData> GetCityDataAsync(string cityName)
{
    // Check if we have cached data
    if (cityDataCache.TryGetValue(cityName, out CityData cachedData))
    {
        // If the data is fresh, return it immediately
        TimeSpan age = DateTime.Now - cachedData.lastUpdated;
        if (age.TotalSeconds < autoRefreshInterval)
        {
            return cachedData;
        }
    }

    try
    {
        // Try to fetch from network
        CityData freshData = await dataService.GetCityDataAsync(cityName);
        if (freshData != null)
        {
            // Update cache
            freshData.lastUpdated = DateTime.Now;
            cityDataCache[cityName] = freshData;

            // Save to persistent cache
            await cacheService.SaveCityDataAsync(freshData);

            // Notify subscribers
            OnCityDataUpdated?.Invoke(cityName);
```

```
                return freshData;
            }
        }
        catch (Exception ex)
        {
            Debug.LogError($"Error fetching data for {cityName}: {ex.Message}");
        }

        // If network failed or returned null, try to get data from persistent cache
        if (cachedData == null)
        {
            cachedData = await cacheService.LoadCityDataAsync(cityName);
            if (cachedData != null)
            {
                cityDataCache[cityName] = cachedData;
            }
        }

        return cachedData;
    }

    public async UniTask<List<string>> GetAllCityNamesAsync()
    {
        try
        {
            // Try to fetch from network
            List<string> cityNames = await dataService.GetAllCityNamesAsync();
            if (cityNames != null && cityNames.Count > 0)
            {
                // Cache the city names
                await cacheService.SaveCityListAsync(cityNames);
                return cityNames;
            }
        }
        catch (Exception ex)
        {
            Debug.LogError($"Error fetching city names: {ex.Message}");
        }

        // If network failed or returned empty list, try cached names
        return await cacheService.LoadCityListAsync();
    }

    public async UniTask<List<HistoricalDataPoint>> GetHistoricalDataAsync(
        string cityName,
        string metricName,
```

```
    DateTime startDate,
    DateTime endDate)
{
    string cacheKey = $"{cityName}_{metricName}_{startDate.ToString("yyyyMMdd")}_{endDa

    // Check if we have cached data
    if (historicalDataCache.TryGetValue(cacheKey, out List<HistoricalDataPoint> cachedD
    {
        return cachedData;
    }

    try
    {
        // Try to fetch from network
        List<HistoricalDataPoint> data = await dataService.GetHistoricalDataAsync(
            cityName, metricName, startDate, endDate);

        if (data != null)
        {
            // Update cache
            historicalDataCache[cacheKey] = data;

            // Save to persistent cache
            await cacheService.SaveHistoricalDataAsync(cityName, metricName, startDate,

            return data;
        }
    }
    catch (Exception ex)
    {
        Debug.LogError($"Error fetching historical data: {ex.Message}");
    }

    // If network failed, try to get data from persistent cache
    return await cacheService.LoadHistoricalDataAsync(cityName, metricName, startDate,
}

public async UniTask RefreshAllData()
{
    // Get all city names
    List<string> cityNames = await GetAllCityNamesAsync();

    // Refresh data for each city
    foreach (string cityName in cityNames)
    {
        try
        {
```

```
                {
                    CityData freshData = await dataService.GetCityDataAsync(cityName);
                    if (freshData != null)
                    {
                        // Update cache
                        freshData.lastUpdated = DateTime.Now;
                        cityDataCache[cityName] = freshData;

                        // Save to persistent cache
                        await cacheService.SaveCityDataAsync(freshData);

                        // Notify subscribers
                        OnCityDataUpdated?.Invoke(cityName);
                    }
                }
                catch (Exception ex)
                {
                    Debug.LogError($"Error refreshing data for {cityName}: {ex.Message}");
                }
            }
        }

        public void ClearCaches()
        {
            cityDataCache.Clear();
            historicalDataCache.Clear();
        }
    }
```

4. Google Sheets Integration

4.1 Setting up Google Sheets API

1. **Create Google Cloud Project**
   - Go to Google Cloud Console
   - Create a new project (e.g., "BSN Dashboard")
   - Enable the Google Sheets API
2. **Set up API credentials**
   - In Google Cloud Console, go to APIs & Services > Credentials
   - Create API Key (for simple access) or OAuth 2.0 Client ID (for secure access)
   - Set appropriate restrictions on the API key
3. **Create a Sheets Service Account**
   - Go to IAM & Admin > Service Accounts
   - Create a new service account
   - Download the JSON key file
   - Store this securely (not in version control)
4. **Set up the Google Sheet**
   - Create a new Google Sheet
   - Set up sheets for:
     - Cities (with metadata)
     - Social Media Metrics
     - Event Metrics
     - Historical Data
     - User Accounts (for admin usage)
   - Share the sheet with your service account

4.2 Creating Google Sheets API Wrapper

1. **Create GoogleSheetsConfig scriptable object**

csharp

```csharp
// GoogleSheetsConfig.cs
using UnityEngine;

[CreateAssetMenu(fileName = "GoogleSheetsConfig", menuName = "BSN/Google Sheets Config")]
public class GoogleSheetsConfig : ScriptableObject
{
    [Header("API Settings")]
    public string apiKey;
    public string spreadsheetId;

    [Header("Sheet Names")]
    public string citiesSheetName = "Cities";
    public string socialMediaSheetName = "SocialMedia";
    public string eventsSheetName = "Events";
    public string historicalDataSheetName = "HistoricalData";
    public string usersSheetName = "Users";

    [Header("Caching Settings")]
    public int cacheDurationMinutes = 5;
}
```

2. **Create GoogleSheetsService class**

csharp

```csharp
// GoogleSheetsService.cs
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using UnityEngine.Networking;
using Newtonsoft.Json;
using Cysharp.Threading.Tasks;

public class GoogleSheetsService : IDataService
{
    private readonly GoogleSheetsConfig config;
    private readonly string baseUrl = "https://sheets.googleapis.com/v4/spreadsheets/";

    public GoogleSheetsService(GoogleSheetsConfig config)
    {
        this.config = config;
    }
```

```
public async Task<CityData> GetCityDataAsync(string cityName)
{
    try
    {
        // Get social media metrics
        var socialMedia = await GetSocialMediaMetricsAsync(cityName);

        // Get event metrics
        var events = await GetEventMetricsAsync(cityName);

        // Get achievements
        var achievements = await GetAchievementsAsync(cityName);

        // Combine into city data
        CityData cityData = new CityData
        {
            cityName = cityName,
            socialMedia = socialMedia,
            events = events,
            achievements = achievements,
            lastUpdated = DateTime.Now
        };

        return cityData;
    }
    catch (Exception ex)
    {
        Debug.LogError($"Error getting city data for {cityName}: {ex.Message}");
        return null;
    }
}

public async Task<List<string>> GetAllCityNamesAsync()
{
    string range = $"{config.citiesSheetName}!A2:A";
    string url = $"{baseUrl}{config.spreadsheetId}/values/{range}?key={config.apiKey}";

    using (UnityWebRequest request = UnityWebRequest.Get(url))
    {
        await request.SendWebRequest();

        if (request.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError($"Error fetching city names: {request.error}");
            return new List<string>();
        }
```

```
        // Parse response
        var response = JsonConvert.DeserializeObject<SheetResponse>(request.downloadHar

        List<string> cityNames = new List<string>();
        if (response.values != null)
        {
            foreach (var row in response.values)
            {
                if (row.Count > 0 && !string.IsNullOrEmpty(row[0]))
                {
                    cityNames.Add(row[0]);
                }
            }
        }

        return cityNames;
    }
}

public async Task<List<HistoricalDataPoint>> GetHistoricalDataAsync(
    string cityName,
    string metricName,
    DateTime startDate,
    DateTime endDate)
{
    string formattedStartDate = startDate.ToString("yyyy-MM-dd");
    string formattedEndDate = endDate.ToString("yyyy-MM-dd");

    // Format: Query for specific city, metric, and date range
    string range = $"{config.historicalDataSheetName}!A2:D";
    string url = $"{baseUrl}{config.spreadsheetId}/values/{range}?key={config.apiKey}";

    using (UnityWebRequest request = UnityWebRequest.Get(url))
    {
        await request.SendWebRequest();

        if (request.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError($"Error fetching historical data: {request.error}");
            return new List<HistoricalDataPoint>();
        }

        // Parse response
        var response = JsonConvert.DeserializeObject<SheetResponse>(request.downloadHar

        List<HistoricalDataPoint> dataPoints = new List<HistoricalDataPoint>();
```

```
        if (response.values != null)
        {
            foreach (var row in response.values)
            {
                if (row.Count >= 4)
                {
                    // Check if row matches our criteria
                    DateTime date = DateTime.Parse(row[0]);
                    string rowCityName = row[1];
                    string rowMetricName = row[2];
                    double value = double.Parse(row[3]);

                    if (rowCityName.Equals(cityName, StringComparison.OrdinalIgnoreCase
                        rowMetricName.Equals(metricName, StringComparison.OrdinalIgnore
                        date >= startDate && date <= endDate)
                    {
                        dataPoints.Add(new HistoricalDataPoint(date, metricName, value,
                    }
                }
            }
        }

        return dataPoints;
    }
}

public async Task<bool> UpdateAchievementProgressAsync(string cityName, string achievem
{
    // In a real implementation, you'd need to use the Google Sheets API to update the
    // This requires OAuth 2.0 authentication, not just an API key

    // This is a simplified version - in reality, you'd use a different API endpoint an
    Debug.Log($"Updating achievement {achievementId} for {cityName} to {newValue}");

    // Simulating success
    return true;
}

private async Task<SocialMediaMetrics> GetSocialMediaMetricsAsync(string cityName)
{
    string range = $"{config.socialMediaSheetName}!A2:D";
    string url = $"{baseUrl}{config.spreadsheetId}/values/{range}?key={config.apiKey}";

    using (UnityWebRequest request = UnityWebRequest.Get(url))
    {
        await request.SendWebRequest();
```

```
        if (request.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError($"Error fetching social media metrics: {request.error}");
            return new SocialMediaMetrics();
        }

        // Parse response
        var response = JsonConvert.DeserializeObject<SheetResponse>(request.downloadHar

        if (response.values != null)
        {
            foreach (var row in response.values)
            {
                if (row.Count >= 4 && row[0].Equals(cityName, StringComparison.OrdinalI
                {
                    return new SocialMediaMetrics(
                        int.Parse(row[1]), // Instagram followers
                        int.Parse(row[2]), // TikTok followers
                        int.Parse(row[3])  // TikTok likes
                    );
                }
            }
        }

        // City not found, return empty metrics
        return new SocialMediaMetrics();
    }
}

private async Task<EventMetrics> GetEventMetricsAsync(string cityName)
{
    string range = $"{config.eventsSheetName}!A2:D";
    string url = $"{baseUrl}{config.spreadsheetId}/values/{range}?key={config.apiKey}";

    using (UnityWebRequest request = UnityWebRequest.Get(url))
    {
        await request.SendWebRequest();

        if (request.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError($"Error fetching event metrics: {request.error}");
            return new EventMetrics();
        }

        // Parse response
        var response = JsonConvert.DeserializeObject<SheetResponse>(request.downloadHar
```

```
        if (response.values != null)
        {
            foreach (var row in response.values)
            {
                if (row.Count >= 4 && row[0].Equals(cityName, StringComparison.OrdinalI
                {
                    return new EventMetrics(
                        int.Parse(row[1]),        // Tickets sold
                        float.Parse(row[2]),      // Average attendance
                        int.Parse(row[3])         // Total events
                    );
                }
            }
        }

        // City not found, return empty metrics
        return new EventMetrics();
    }
}

private async Task<List<Achievement>> GetAchievementsAsync(
```