UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

XWifi Android App

Anul IV, Tehnologia Informației

Autor: Bildea Ioan-Cristian

Grupa: 30644

FACULTATEA DE AUTOMATICĂ
ȘI CALCULATOARE

Cuprins

Let me create the remaining sections for your document:

“‘latex

1 Technologies Used

1.1 Mobile Application Stack

The XWiFi mobile application is built using the following technologies:

- **Flutter 3.x:** Google’s UI toolkit for building natively compiled applications for mobile, web, and desktop from a single codebase. Flutter provides:
  - Fast development through hot reload
  - Expressive and flexible UI
  - Native performance
  - Access to platform features via plugins
- **Dart 3.x:** A client-optimized language for fast apps on any platform. Dart offers:
  - Type safety
  - Asynchronous programming with async/await
  - Garbage collection
  - Rich standard library
- **Provider Pattern:** A state management approach that:
  - Separates business logic from UI
  - Provides an efficient way to propagate changes
  - Simplifies testing through dependency injection

1.2 Backend Stack

The XWiFi backend is built using the following technologies:

- **.NET 8:** Microsoft’s cross-platform framework for building modern applications. .NET 8 offers:
  - High performance
  - Minimal API design for reduced boilerplate
  - Strong type system via C# 12
  - Integrated dependency injection
- **Entity Framework Core 9.x:** Microsoft’s modern object-database mapper for .NET that:
  - Enables developers to work with a database using .NET objects
  - Supports a variety of database engines
  - Provides database migrations for schema evolution
  - Offers a LINQ-based query API
- **PostgreSQL:** A powerful, open-source object-relational database system that:
  - Provides robust data integrity
  - Offers excellent performance for both simple and complex queries
  - Supports advanced data types
  - Ensures ACID compliance
- **Minimal API:** A simplified approach for building HTTP APIs in .NET that:

  - Reduces ceremony and boilerplate
  - Integrates seamlessly with dependency injection
  - Produces cleaner, more focused code
  - Delivers excellent performance

1.3 Development Environment

The development environment consists of:

- **Intellij + Android Emulator in the Intellij:** For Flutter and Dart development and emulator
- **Rider:** For .NET backend development
- **Android Studio/Emulator:** For Android testing
- **PowerShell:** For running commands and scripts

2 Key Packages and Dependencies

2.1 Flutter Application Packages

The XWiFi mobile application relies on several key packages:

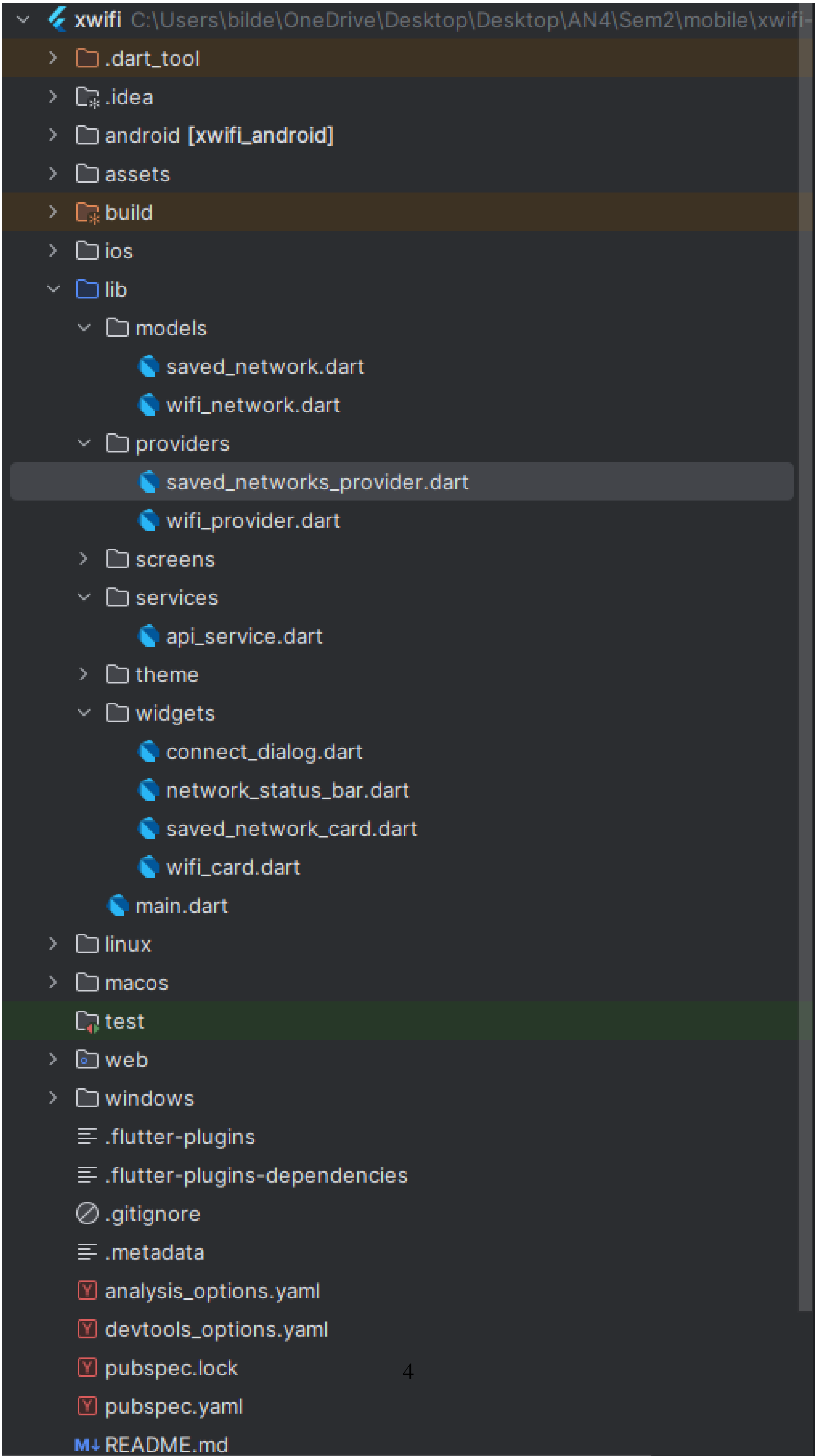
2.2 .NET Backend Packages

The backend API relies on the following NuGet packages:

3 Project Structure and Components

3.1 Flutter Application Structure

The Flutter application follows a modular architecture organized into the following directory structure:



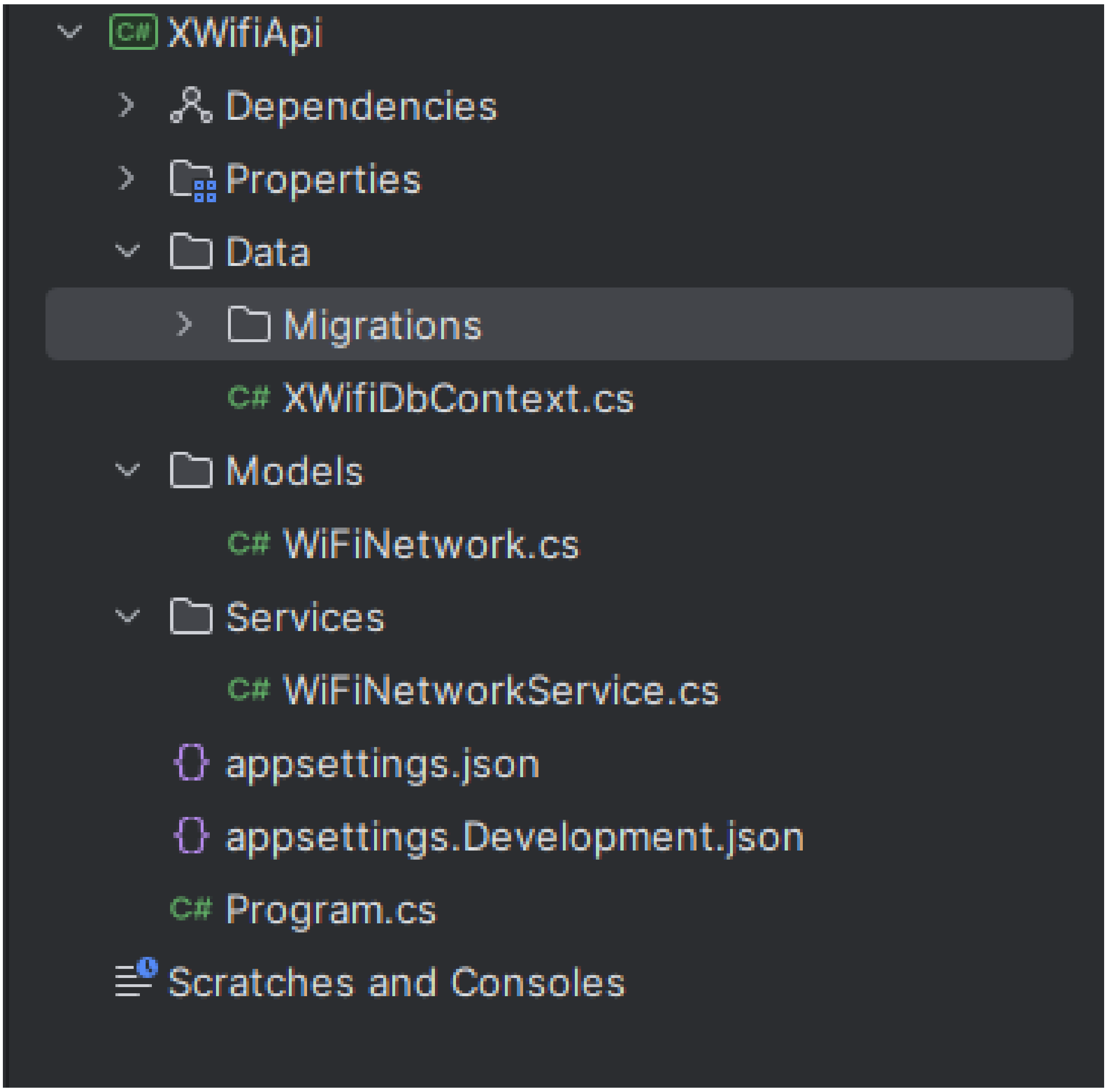
```
xwifi C:\Users\bilde\OneDrive\Desktop\Desktop\AN4\Sem2\mobile\xwifi-
  .dart_tool
  .idea
  android [xwifi_android]
  assets
  build
  ios
  lib
    models
      saved_network.dart
      wifi_network.dart
    providers
      saved_networks_provider.dart
      wifi_provider.dart
    screens
    services
      api_service.dart
    theme
    widgets
      connect_dialog.dart
      network_status_bar.dart
      saved_network_card.dart
      wifi_card.dart
    main.dart
  linux
  macos
  test
  web
  windows
  .flutter-plugins
  .flutter-plugins-dependencies
  .gitignore
  .metadata
  analysis_options.yaml
  devtools_options.yaml
  pubspec.lock
  pubspec.yaml
  README.md
```

Each component has specific responsibilities:

- **Models**: Encapsulate the data structures and business logic related to WiFi networks
  - `wifi_network.dart`: Represents a WiFi network with properties for SSID, signal strength, and security
  - `saved_network.dart`: Extends network information with password and personal notes
- **Providers**: Implement state management using the Provider pattern
  - `wifi_provider.dart`: Manages WiFi scanning, connectivity, and device-level operations
  - `saved_networks_provider.dart`: Handles retrieving, storing, and deleting saved networks
- **Screens**: Define the UI for different application features
  - `home_screen.dart`: Primary screen for scanning and connecting to networks
  - `details_screen.dart`: Displays technical details about the connected network
  - `share_screen.dart`: Interface for saving and sharing network credentials
  - `settings_screen.dart`: Application configuration and preferences
- **Services**: Handle external communication
  - `api_service.dart`: Manages communication with the backend API
- **Widgets**: Provide reusable UI components
  - `wifi_card.dart`: Displays a WiFi network with signal strength and security info
  - `saved_network_card.dart`: Shows saved network with options to connect or delete
  - `connect_dialog.dart`: Dialog for entering WiFi password
  - `network_status_bar.dart`: Shows current connection information

3.2 .NET Backend Structure

The .NET backend follows a clean architecture with the following structure:



```
XWifiApi
  Dependencies
  Properties
  Data
    Migrations
    XWifiDbContext.cs
  Models
    WiFiNetwork.cs
  Services
    WiFiNetworkService.cs
  appsettings.json
  appsettings.Development.json
  Program.cs
Scratches and Consoles
```

Figura 2: .NET app

The backend components have the following responsibilities:

- **Program.cs**: Configures the application, dependency injection, and defines API endpoints
- **Models/WiFiNetwork.cs**: Defines the data structure for WiFi networks with:
  - Network identifiers (ID, SSID)
  - Optional password
  - Security capabilities
  - User notes
  - Creation timestamp
- **Data/XWifiDbContext.cs**: Configures Entity Framework Core with:
  - Entity configurations
  - PostgreSQL data type mappings
  - DateTime handling for UTC conversion
- **Services/WiFiNetworkService.cs**: Implements business logic for:
  - Retrieving all networks

  - Getting a network by ID
  - Adding a new network
  - Deleting a network

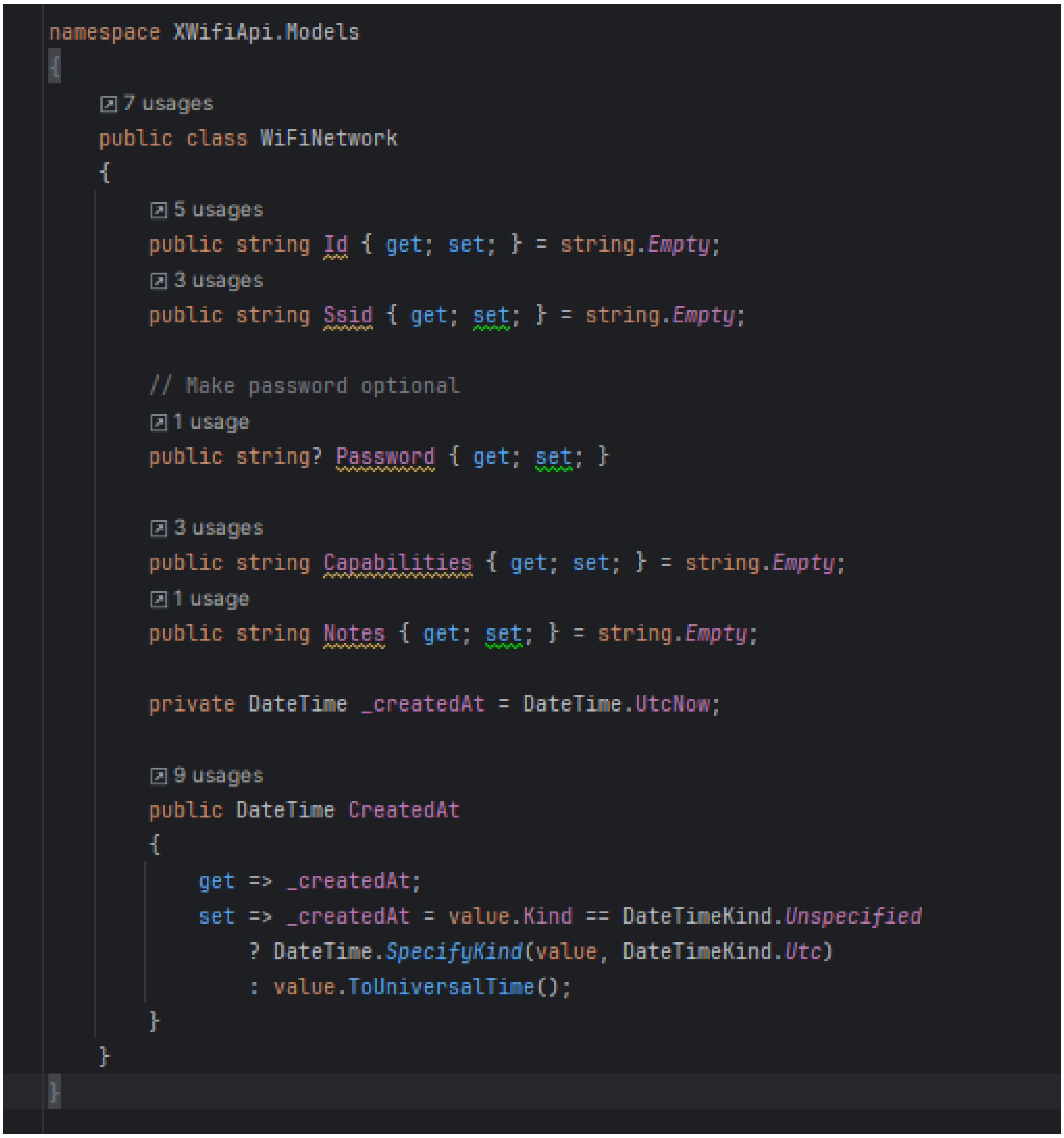
4 API Endpoints and Communication

4.1 API Endpoints

The .NET backend exposes the following RESTful endpoints:

4.2 Data Models

The primary data model for API communication is the WiFiNetwork object:



```
namespace XWifiApi.Models
{
    7 usages
    public class WiFiNetwork
    {
        5 usages
        public string Id { get; set; } = string.Empty;
        3 usages
        public string Ssid { get; set; } = string.Empty;

        // Make password optional
        1 usage
        public string? Password { get; set; }

        3 usages
        public string Capabilities { get; set; } = string.Empty;
        1 usage
        public string Notes { get; set; } = string.Empty;

        private DateTime _createdAt = DateTime.UtcNow;

        9 usages
        public DateTime CreatedAt
        {
            get => _createdAt;
            set => _createdAt = value.Kind == DateTimeKind.Unspecified
                ? DateTime.SpecifyKind(value, DateTimeKind.Utc)
                : value.ToUniversalTime();
        }
    }
}
```

Figura 3: .NET app

4.3 Client-Server Communication

The Flutter application communicates with the backend using the `ApiService` class, which:

- Handles HTTP requests using the `http` package
- Serializes and deserializes JSON data
- Manages error handling and retries
- Provides an abstraction layer for the UI components

Example of communication flow:

1. User saves a WiFi network on the Share screen
2. `SavedNetworksProvider` calls `ApiService.saveNetwork()`
3. `ApiService` serializes the network object to JSON

4. A POST request is sent to `/api/networks`
5. The .NET backend processes the request and saves to PostgreSQL
6. A response with the saved network data is returned
7. `ApiService` deserializes the response to a `SavedNetwork` object
8. `SavedNetworksProvider` updates its state with the new network
9. The UI refreshes to show the newly saved network

5 Flutter UI Components

5.1 Screen Structure

The XWiFi application consists of four main screens:

- **Home Screen (Scanner)**: The primary interface for discovering WiFi networks
  - Displays a list of available networks with signal strength
  - Provides a refresh button to rescan for networks
  - Shows the currently connected network
  - Allows users to connect to networks by tapping on them
- **Details Screen**: Shows technical information about the connected network
  - Displays IP address, subnet mask, and gateway
  - Shows signal strength, frequency, and channel
  - Provides BSSID and security type information
  - Offers options to disconnect or forget the network
- **Share Screen**: Interface for saving and sharing WiFi credentials
  - Shows the currently connected network
  - Allows users to save networks with or without passwords
  - Displays a list of previously saved networks
  - Provides options to connect to or delete saved networks
- **Settings Screen**: Application configuration
  - Offers auto-scan interval configuration
  - Provides options to enable/disable WiFi
  - Shows application information and version
  - Contains links to help and documentation

5.2 Key UI Components

The application uses several reusable widgets to maintain consistency:

- **WiFi Card**: Displays a scanned WiFi network with:
  - Network name and security type
  - Signal strength indicator
  - Connect button
- **Saved Network Card**: Shows a saved network with:
  - Network name and security type
  - Password visibility toggle
  - Connect and delete options

  - Notes display
- **Connect Dialog**: Modal dialog for connecting to a network:
  - Password input field with visibility toggle
  - Connect and cancel buttons
  - Auto-fill for saved networks
- **Network Status Bar**: Shows current connection information:
  - Connected network name
  - Signal strength icon
  - IP address

5.3 Navigation

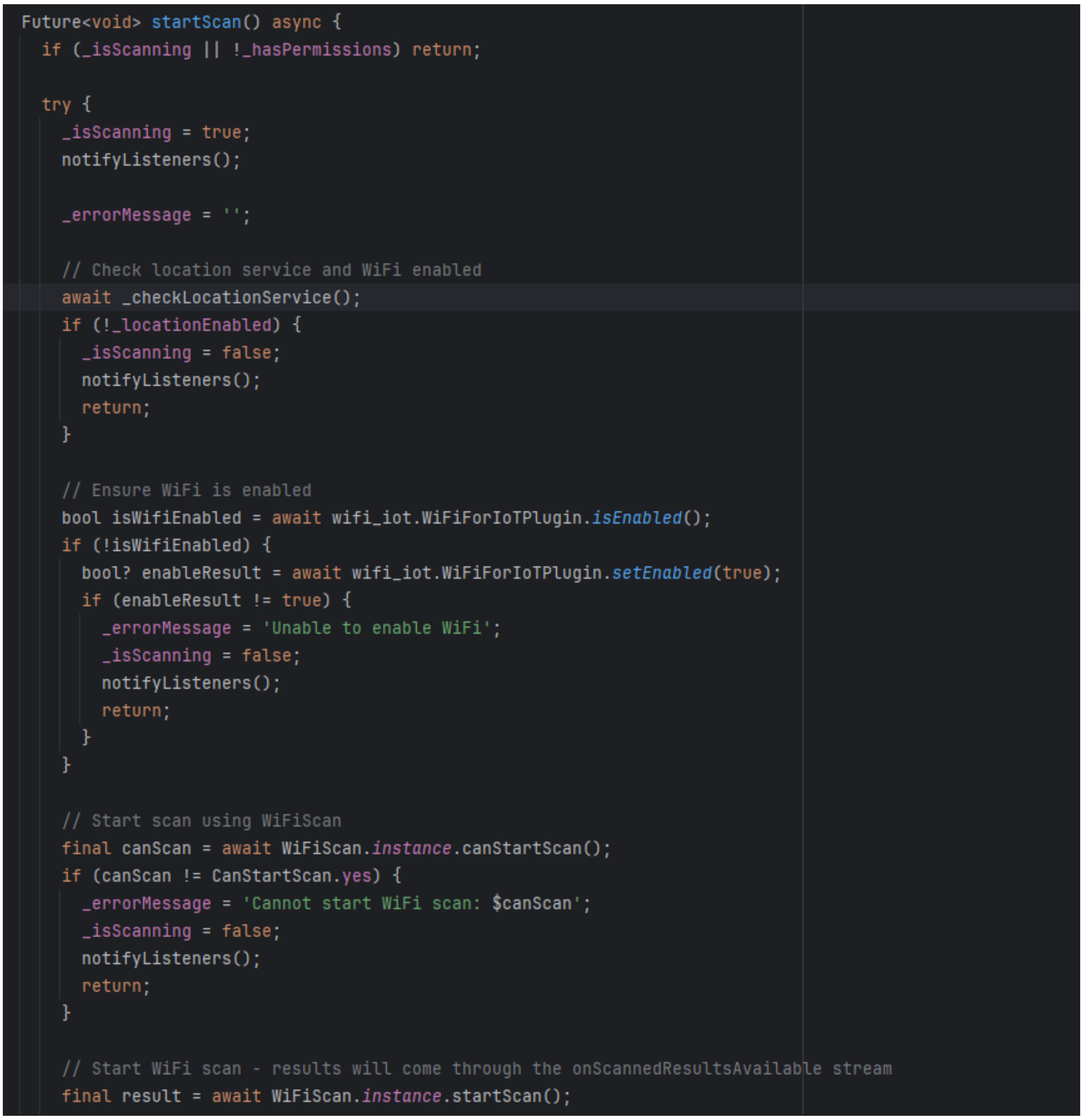
The application uses a bottom navigation bar with four destinations corresponding to the main screens. This provides:

- Easy access to all main features
- Persistent navigation across the application
- Visual indicators of the current section

6 Implementation Details

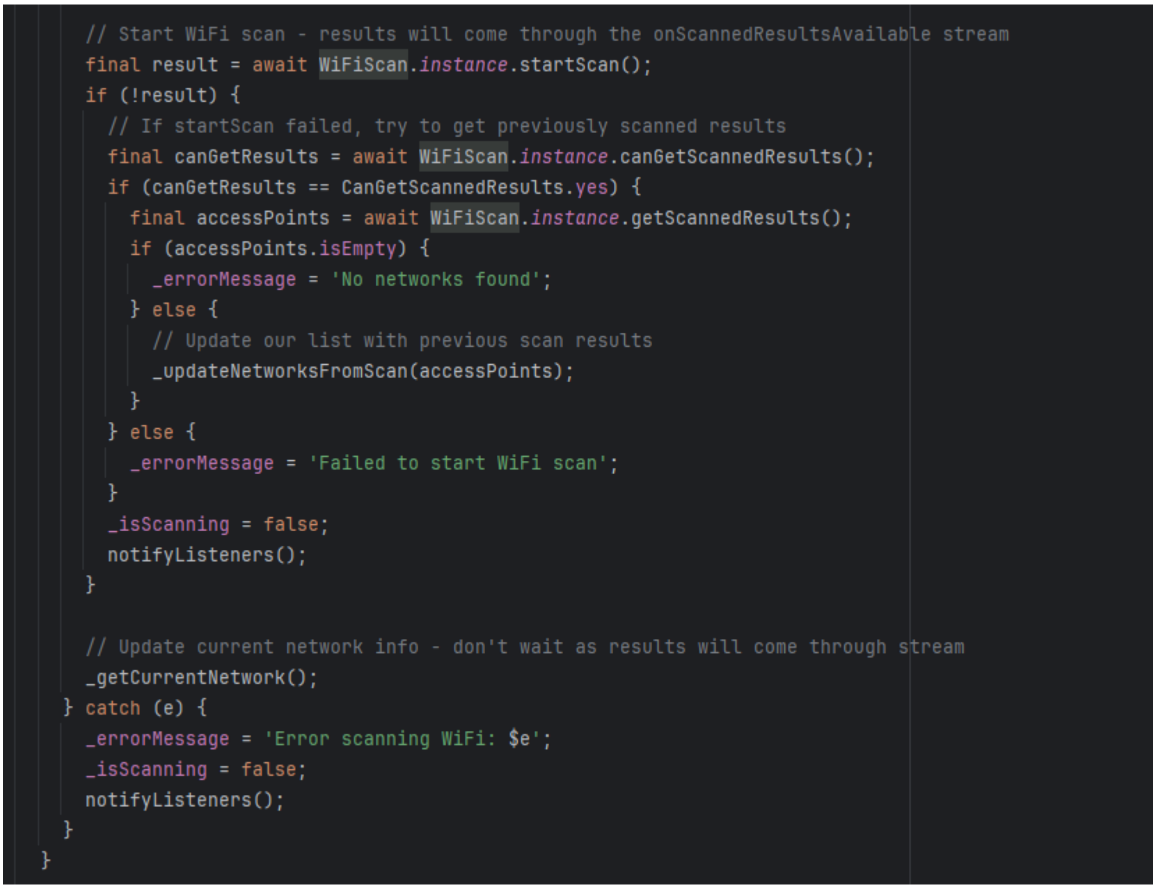
6.1 WiFi Scanning Implementation

The WiFi scanning functionality is implemented in the `WifiProvider` class:



```
Future<void> startScan() async {
  if (_isScanning || !_hasPermissions) return;

  try {
    _isScanning = true;
    notifyListeners();

    _errorMessage = '';

    // Check location service and WiFi enabled
    await _checkLocationService();
    if (!_locationEnabled) {
      _isScanning = false;
      notifyListeners();
      return;
    }

    // Ensure WiFi is enabled
    bool isWifiEnabled = await wifi_iot.WiFiForIoTPlugin.isEnabled();
    if (!isWifiEnabled) {
      bool? enableResult = await wifi_iot.WiFiForIoTPlugin.setEnabled(true);
      if (enableResult != true) {
        _errorMessage = 'Unable to enable WiFi';
        _isScanning = false;
        notifyListeners();
        return;
      }
    }

    // Start scan using WiFiScan
    final canScan = await WiFiScan.instance.canStartScan();
    if (canScan != CanStartScan.yes) {
      _errorMessage = 'Cannot start WiFi scan: $canScan';
      _isScanning = false;
      notifyListeners();
      return;
    }

    // Start WiFi scan - results will come through the onScannedResultsAvailable stream
    final result = await WiFiScan.instance.startScan();
```

Figura 4: start$_scan$



```
    // Start WiFi scan - results will come through the onScannedResultsAvailable stream
    final result = await WiFiScan.instance.startScan();
    if (!result) {
      // If startScan failed, try to get previously scanned results
      final canGetResults = await WiFiScan.instance.canGetScannedResults();
      if (canGetResults == CanGetScannedResults.yes) {
        final accessPoints = await WiFiScan.instance.getScannedResults();
        if (accessPoints.isEmpty) {
          _errorMessage = 'No networks found';
        } else {
          // Update our list with previous scan results
          _updateNetworksFromScan(accessPoints);
        }
      } else {
        _errorMessage = 'Failed to start WiFi scan';
      }
      _isScanning = false;
      notifyListeners();
    }

    // Update current network info - don't wait as results will come through stream
    _getCurrentNetwork();
  } catch (e) {
    _errorMessage = 'Error scanning WiFi: $e';
    _isScanning = false;
    notifyListeners();
  }
}
```

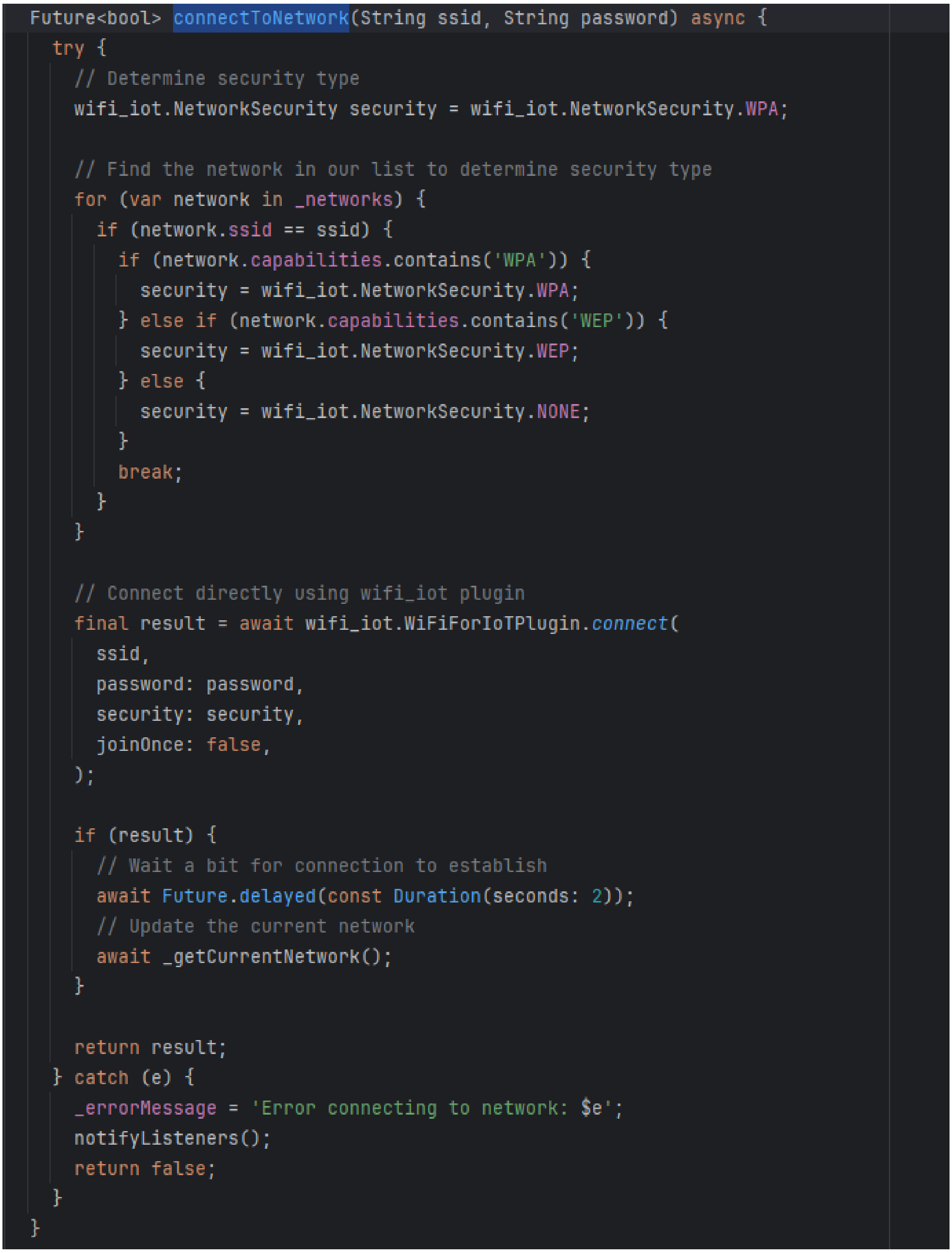
Figura 5: start$_{s}can$

Key aspects of this implementation:

- Permission checks before scanning
- Automatic enabling of WiFi if disabled
- Error handling for various failure scenarios
- Stream-based results processing
- State management with notifyListeners()

6.2 Network Connection Implementation

Connecting to a WiFi network is handled by the `connectToNetwork` method:

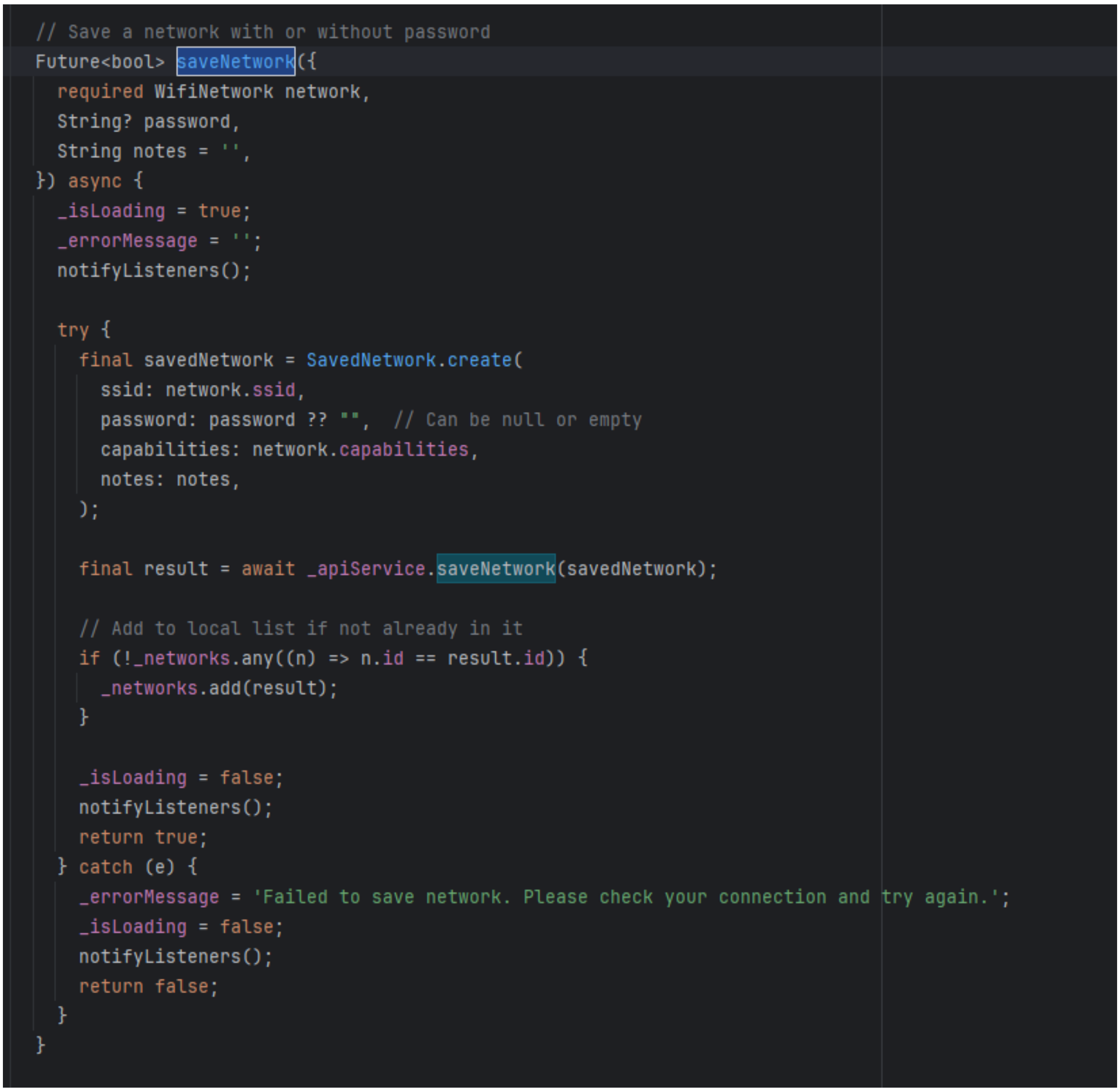


```
Future<bool> connectToNetwork(String ssid, String password) async {
  try {
    // Determine security type
    wifi_iot.NetworkSecurity security = wifi_iot.NetworkSecurity.WPA;

    // Find the network in our list to determine security type
    for (var network in _networks) {
      if (network.ssid == ssid) {
        if (network.capabilities.contains('WPA')) {
          security = wifi_iot.NetworkSecurity.WPA;
        } else if (network.capabilities.contains('WEP')) {
          security = wifi_iot.NetworkSecurity.WEP;
        } else {
          security = wifi_iot.NetworkSecurity.NONE;
        }
        break;
      }
    }

    // Connect directly using wifi_iot plugin
    final result = await wifi_iot.WiFiForIoTPlugin.connect(
      ssid,
      password: password,
      security: security,
      joinOnce: false,
    );

    if (result) {
      // Wait a bit for connection to establish
      await Future.delayed(const Duration(seconds: 2));
      // Update the current network
      await _getCurrentNetwork();
    }

    return result;
  } catch (e) {
    _errorMessage = 'Error connecting to network: $e';
    notifyListeners();
    return false;
  }
}
```

Figura 6: connectToNetwork

6.3 Network Saving Implementation

Saving a network is handled by the `SavedNetworksProvider`:



```
// Save a network with or without password
Future<bool> saveNetwork({
  required WifiNetwork network,
  String? password,
  String notes = '',
}) async {
  _isLoading = true;
  _errorMessage = '';
  notifyListeners();

  try {
    final savedNetwork = SavedNetwork.create(
      ssid: network.ssid,
      password: password ?? "",  // Can be null or empty
      capabilities: network.capabilities,
      notes: notes,
    );

    final result = await _apiService.saveNetwork(savedNetwork);

    // Add to local list if not already in it
    if (!_networks.any((n) => n.id == result.id)) {
      _networks.add(result);
    }

    _isLoading = false;
    notifyListeners();
    return true;
  } catch (e) {
    _errorMessage = 'Failed to save network. Please check your connection and try again.';
    _isLoading = false;
    notifyListeners();
    return false;
  }
}
```

Figura 7: saveNetwork

6.4 Backend Service Implementation

The backend service implements the core business logic: It is used a minimal API with .NET SDK 8.

7 Steps to Recreate the Project

7.1 Prerequisites

Before starting, ensure you have the following tools installed:

- Flutter SDK (latest stable version)
- .NET 8 SDK
- Intelijji with Flutter/Dart extensions
- Rider 2022 (for backend development)

- PostgreSQL database server

8 Flutter Application Setup

- **Create a new Flutter project:** [language=bash] flutter create xwifi cd xwifi
- **Add dependencies** (in `pubspec.yaml` under `dependencies`):
  - `flutter`: SDK

  - `wifi_scan`: `0̂.4.1+2`
  - `wifi_iot`: `0̂.3.19+2`
  - `network_info_plus`: `6̂.1.4`
  - `permission_handler`: `1̂1.0.0`
  - `http`: `0̂.13.5`
  - `provider`: `6̂.1.1`
  - `cupertino_icons`: `1̂.0.8`
- **Create directory structure:** [language=bash] mkdir -p lib/models lib/providers lib/screens lib/services lib/widgets
- **Implement models:**
  - `lib/models/wifi_network.dart`
  - `lib/models/saved_network.dart`
- **Implement services:**
  - `lib/services/api_service.dart`
- **Implement providers:**
  - `lib/providers/wifi_provider.dart`
  - `lib/providers/saved_networks_provider.dart`
- **Implement widgets:**
  - `lib/widgets/wifi_card.dart`
  - `lib/widgets/saved_network_card.dart`
  - `lib/widgets/connect_dialog.dart`
  - `lib/widgets/network_status_bar.dart`
- **Implement screens:**
  - `lib/screens/home_screen.dart`
  - `lib/screens/details_screen.dart`
  - `lib/screens/share_screen.dart`
  - `lib/screens/settings_screen.dart`
- **Update entry point and providers** in `lib/main.dart`.
- **Configure Android permissions** (in `android/app/src/main/AndroidManifest.xml`):
  - `<uses-permission android:name="android.permission.INTERNET"/>`
  - `<uses-permission android:name="android.permission.ACCESS_NETWORK_S`
  - `<uses-permission android:name="android.permission.ACCESS_WIFI_STAT`
  - `<uses-permission android:name="android.permission.CHANGE_WIFI_STAT`
  - `<uses-permission android:name="android.permission.ACCESS_FINE_LOCA`
  - `<uses-permission android:name="android.permission.ACCESS_COARSE_LO`

9 .NET Backend Setup

- **Create a new .NET minimal API project:** [language=bash] mkdir -p server cd server dotnet new web -n XWifiApi cd XWifiApi
- **Add NuGet packages:**
- **Create directory structure:** [language=bash] mkdir -p Models Data Services
- **Define WiFiNetwork model** in `Models/WiFiNetwork.cs`:
- **Configure DbContext** in `Data/XWifiDbContext.cs`:
- **Add WiFiNetworkService** in `Services/WiFiNetworkService.cs`.
- **Configure API endpoints** in `Program.cs`.
- **Set connection string** in `appsettings.json`:
- **Migrate database**

10 Running the Application

- **Start backend server:**
- **Configure Flutter API URL** in `lib/main.dart`:
- **Run Flutter app:**

11 Future Enhancements

11.1 Potential Improvements

- QR code sharing for networks
- User authentication and per-user collections
- Automatic reconnection by location
- Integrated network speed tests
- Security analysis of connected networks
- Desktop (Windows/macOS/Linux) support
- Cross-device data synchronization
- Advanced filtering and sorting
- Historical analytics and usage statistics

11.2 Technical Improvements

- Offline caching of saved networks
- Encryption of stored credentials
- UI and scan performance optimizations
- Unit, integration, and UI test coverage
- CI/CD pipeline setup
- Localization for multiple languages
- Accessibility enhancements

12 Conclusion

The XWiFi app demonstrates a modular, cross-platform solution for WiFi management, using:

- Clean architecture and separation of concerns
- Provider-based state management
- RESTful .NET minimal APIs with PostgreSQL
- Platform integrations for scanning and connectivity

It's designed for easy extension and maintenance as new features and platforms arise.

Package	Version	Purpose and Implementation Details
wifi_scan	0.4.1+2	Provides WiFi scanning functionality across platforms. Used in WifiProvider to: • Discover available WiFi networks • Retrieve signal strength information • Get network capabilities (security type) • Monitor scan results via stream subscription
wifi_iot	0.3.19+2	Enables direct interaction with the device's WiFi. Used in WifiProvider to: • Connect to WiFi networks with credentials • Disconnect from networks • Get current frequency and signal strength • Retrieve advanced network information
network_info_plus	6.1.4	Retrieves detailed network connection information. Used in WifiProvider to: • Get WiFi name (SSID) and BSSID • Retrieve IP address and IPv6 information • Get subnet mask, gateway IP, and broadcast address
permission_handler	11.0.0	Manages runtime permissions. Used in WifiProvider to: • Request location permissions (required for WiFi scanning on Android) • Check permission status • Handle permission callbacks
http	0.13.5	Provides HTTP client functionality. Used in ApiService to: • Make GET requests to retrieve saved networks • Send POST requests to save networks • Issue DELETE requests to remove networks • Handle JSON serialization/deserialization
provider	6.1.1	Implements state management. Used throughout the application to: • Manage WiFi scanning state via WifiProvider • Handle saved networks state via SavedNetworksProvider • Propagate state changes to UI components • Implement dependency injection

Package	Version	Usage Details
`Npgsql.EntityFrameworkCore.PostgreSQL`	9.0.4	• Connects Entity Framework Core to PostgreSQL • Handles data type conversions • Manages connection pooling • Implements PostgreSQL-specific features
`Microsoft.EntityFrameworkCore.Design`	9.0.4	• Provides design-time components for EF Core tools • Enables database migrations • Supports scaffolding of models from existing databases • Facilitates database schema evolution
`Swashbuckle.AspNetCore`	8.1.1	• Generates OpenAPI documentation • Provides the Swagger UI for testing endpoints • Enables API exploration during development • Facilitates client code generation

Tabela 2: .NET Backend Dependencies

Endpoint	Method	Description
`/api/networks`	GET	Retrieves all saved WiFi networks. Returns an array of network objects ordered by creation date (newest first).
`/api/networks/{id}`	GET	Retrieves a specific network by its ID. Returns a single network object or 404 Not Found if the network doesn't exist.
`/api/networks`	POST	Creates a new WiFi network entry. Accepts a JSON object with network details including optional password. Returns the created network with a 201 Created status.
`/api/networks/{id}`	DELETE	Deletes a specific network by its ID. Returns 204 No Content on success or 404 Not Found if the network doesn't exist.

Tabela 3: API Endpoints