Perfect! I can see your current architecture and how to best modify it for the remote API setup. Your code is well-structured, which will make the transition smooth.

**Analysis of Your Current Setup**

Your project structure is excellent:

* **MMPD** (MAUI app) - Currently using embedded SQLite
* **MMPD.Data** - EF Core models and context
* **MMPD.Shared** - Blazor components

I can see you have 4 location types (Corporate=1, Metal Mart=2, Service Center=3, Plant=4) and complex hierarchical data with Locations → Departments → Employees.

**Recommended Migration Plan**

**1. Create New Web API Project**

Add a new project to your solution:

<!-- MMPD.Api/MMPD.Api.csproj -->

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>net9.0</TargetFramework>

<Nullable>enable</Nullable>

<ImplicitUsings>enable</ImplicitUsings>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="9.0.5" />

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="9.0.5" />

</ItemGroup>

<ItemGroup>

<ProjectReference Include="..\MMPD.Data\MMPD.Data.csproj" />

</ItemGroup>

</Project>

**2. Update MMPD.Data for SQL Server**

Modify your MMPD.Data.csproj to support both SQLite and SQL Server:

<ItemGroup>

<PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.5" />

<PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="9.0.5" />

<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="9.0.5" />

<PackageReference Include="Newtonsoft.Json" Version="13.0.3" />

</ItemGroup>

**3. API Controllers Based on Your ExportData Class**

// MMPD.Api/Controllers/DirectoryController.cs

[ApiController]

[Route("api/[controller]")]

public class DirectoryController : ControllerBase

{

private readonly AppDbContext \_context;

private readonly ExportData \_exportData;

public DirectoryController(AppDbContext context)

{

\_context = context;

\_exportData = new ExportData(context);

}

// Full sync endpoint for MAUI apps

[HttpGet("sync")]

public async Task<ActionResult<DirectorySyncResponse>> GetDirectorySync(

[FromQuery] string apiKey)

{

if (!IsValidApiKey(apiKey))

return Unauthorized();

try

{

// Use your existing ExportData logic

var jsonData = await \_exportData.GenerateJson();

return Ok(new DirectorySyncResponse

{

Data = jsonData,

SyncTimestamp = DateTime.UtcNow

});

}

catch (Exception ex)

{

return StatusCode(500, $"Sync failed: {ex.Message}");

}

}

// CRUD endpoints for your web admin app

[HttpGet("employees")]

public async Task<ActionResult<List<Employee>>> GetEmployees([FromQuery] string apiKey)

{

if (!IsValidApiKey(apiKey)) return Unauthorized();

return await \_context.Employees

.Include(e => e.EmpDepartment)

.Include(e => e.EmpLocation)

.Where(e => e.Active)

.ToListAsync();

}

[HttpPost("employees")]

public async Task<ActionResult<Employee>> CreateEmployee([FromBody] Employee employee, [FromQuery] string apiKey)

{

if (!IsValidApiKey(apiKey)) return Unauthorized();

\_context.Employees.Add(employee);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetEmployees), new { id = employee.Id }, employee);

}

[HttpPut("employees/{id}")]

public async Task<IActionResult> UpdateEmployee(int id, [FromBody] Employee employee, [FromQuery] string apiKey)

{

if (!IsValidApiKey(apiKey)) return Unauthorized();

if (id != employee.Id) return BadRequest();

\_context.Entry(employee).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return NoContent();

}

[HttpDelete("employees/{id}")]

public async Task<IActionResult> DeleteEmployee(int id, [FromQuery] string apiKey)

{

if (!IsValidApiKey(apiKey)) return Unauthorized();

var employee = await \_context.Employees.FindAsync(id);

if (employee == null) return NotFound();

// Soft delete

employee.Active = false;

await \_context.SaveChangesAsync();

return NoContent();

}

// Similar endpoints for Locations and Departments...

private bool IsValidApiKey(string apiKey)

{

var validKeys = new[] { "maui-app-key-2024", "crud-web-app-key-2024" };

return validKeys.Contains(apiKey);

}

}

public class DirectorySyncResponse

{

public Dictionary<string, object> Data { get; set; } = new();

public DateTime SyncTimestamp { get; set; }

}

**4. API Program.cs**

// MMPD.Api/Program.cs

using Microsoft.EntityFrameworkCore;

using MMPD.Data.Context;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddCors(options =>

{

options.AddPolicy("CompanyPolicy", policy =>

{

policy.WithOrigins("https://yourcompany.com", "https://\*.mcelroy.local")

.AllowAnyHeader()

.AllowAnyMethod();

});

});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseCors("CompanyPolicy");

app.MapControllers();

app.Run();

**5. Update MAUI App for Sync**

Create a new sync service:

// MMPD/Services/SyncService.cs

public class SyncService

{

private readonly HttpClient \_httpClient;

private readonly string \_apiKey = "maui-app-key-2024";

private readonly string \_baseUrl = "https://api.mcelroy.local"; // Your API URL

public SyncService()

{

\_httpClient = new HttpClient { BaseAddress = new Uri(\_baseUrl) };

}

public async Task<bool> SyncDirectoryAsync()

{

try

{

var response = await \_httpClient.GetAsync($"api/directory/sync?apiKey={\_apiKey}");

if (response.IsSuccessStatusCode)

{

var syncData = await response.Content.ReadFromJsonAsync<DirectorySyncResponse>();

// Save to local SQLite using your existing structure

await SaveToLocalDatabase(syncData!.Data);

await SetLastSyncTime(syncData.SyncTimestamp);

return true;

}

}

catch (Exception ex)

{

System.Diagnostics.Debug.WriteLine($"Sync failed: {ex.Message}");

}

return false;

}

private async Task SaveToLocalDatabase(Dictionary<string, object> data)

{

// Use your existing database context to save the synced data

// This replaces your current SeedDatabase method

var dbPath = Path.Combine(FileSystem.AppDataDirectory, "mcelroy\_directory.db");

var options = new DbContextOptionsBuilder<AppDbContext>()

.UseSqlite($"Data Source={dbPath}")

.Options;

using var context = new AppDbContext(options);

await context.Database.EnsureCreatedAsync();

// Parse and save the data (you can adapt your ExportData logic for this)

// Since your data structure is complex, you might want to create a method

// to parse the JSON back into your entities

}

private async Task SetLastSyncTime(DateTime syncTime)

{

await SecureStorage.SetAsync("LastSyncTime", syncTime.ToString("O"));

}

public async Task<DateTime?> GetLastSyncTime()

{

var syncTimeString = await SecureStorage.GetAsync("LastSyncTime");

if (DateTime.TryParse(syncTimeString, out var syncTime))

return syncTime;

return null;

}

}

**6. Update MauiProgram.cs**

public static class MauiProgram

{

public static MauiApp CreateMauiApp()

{

var builder = MauiApp.CreateBuilder();

builder

.UseMauiApp<App>()

.ConfigureFonts(fonts =>

{

fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");

});

// ... your existing configuration ...

// Add sync service

builder.Services.AddSingleton<SyncService>();

// Keep local SQLite for offline capability

var dbPath = Path.Combine(FileSystem.AppDataDirectory, "mcelroy\_directory.db");

builder.Services.AddDbContext<AppDbContext>(options =>

{

options.UseSqlite($"Data Source={dbPath}");

});

// Your existing services...

builder.Services.AddScoped<IDirectoryService, DirectoryService>();

return builder.Build();

}

}

**7. Update App.xaml.cs for Background Sync**

public partial class App : Application

{

private readonly SyncService \_syncService;

public App(SyncService syncService)

{

\_syncService = syncService;

InitializeComponent();

}

protected override Window CreateWindow(IActivationState? activationState)

{

var appWindow = new Window(new MainPage())

{

Title = "McElroy Directory",

// ... your existing window configuration ...

};

// Start background sync

Task.Run(async () =>

{

await InitializeAndSync();

});

return appWindow;

}

private async Task InitializeAndSync()

{

try

{

// Ensure local database exists first

await EnsureLocalDatabaseExists();

// Try to sync with server

var syncSuccessful = await \_syncService.SyncDirectoryAsync();

if (syncSuccessful)

{

System.Diagnostics.Debug.WriteLine("Directory sync successful");

// Notify UI that data was updated

MessagingCenter.Send(this, "DirectoryUpdated");

}

else

{

System.Diagnostics.Debug.WriteLine("Directory sync failed, using cached data");

}

}

catch (Exception ex)

{

System.Diagnostics.Debug.WriteLine($"Sync initialization failed: {ex.Message}");

}

}

private async Task EnsureLocalDatabaseExists()

{

var dbPath = Path.Combine(FileSystem.AppDataDirectory, "mcelroy\_directory.db");

if (!File.Exists(dbPath))

{

// Copy your seed database as fallback

using var dbStream = await FileSystem.OpenAppPackageFileAsync("Data/mcelroy\_directory.db");

using var fileStream = new FileStream(dbPath, FileMode.Create, FileAccess.Write);

await dbStream.CopyToAsync(fileStream);

}

}

}

**Migration Steps**

1. **Create the API project** and test locally
2. **Migrate SQLite data to SQL Server** using EF migrations
3. **Deploy API to your corporate server** with IIS
4. **Update MAUI app** to use sync service
5. **Test offline capability** by disconnecting network

Would you like me to show you the specific migration commands for moving your SQLite data to SQL Server, or would you prefer to see how to modify your existing Blazor components to call the API instead of direct database access?

Also, I notice you have a DirectoryService referenced but not shown - I can help integrate the API calls into that service to minimize changes to your existing Blazor components.