



# Setting Up the Databases in Visual Studio for Dwellingly | AI

## Setting Up the Databases in Visual Studio for Dwellingly | AI

This guide provides detailed instructions to set up the databases for the Dwellingly | AI application, including configuring Entity Framework Core, setting up migrations, and initializing the databases with seed data.

### Step-by-Step Guide

#### Step 1: Define the Database Context and Models

##### 1. Create the Database Context:

- Add a new class `NexHomeAgentDbContext.cs` in the `Dwellingly.Data` project:

```
csharpCopy code
using Microsoft.EntityFrameworkCore;
using NexHomeAgent.Models;

namespace NexHomeAgent.Data
{
```

```

public class NexHomeAgentDbContext : DbContext
{
    public NexHomeAgentDbContext(DbContextOptions<NexHomeAgentDbContext> options)
        : base(options)
    {

        public DbSet<User> Users { get; set; }
        public DbSet<Property> Properties { get; set; }
        public DbSet<Favorite> Favorites { get; set; }
        public DbSet<PropertyImage> PropertyImages { get; set; }
        public DbSet<PropertyHistory> PropertyHistories { get; set; }
    }
}

```

## 2. Define the Models:

- Add the following model classes in the `Dwellingly.Data` project:

- `User.cs`:

```

csharpCopy code
namespace NexHomeAgent.Models
{
    public class User
    {
        public int Id { get; set; }
        public string Email { get; set; }
        public string Password { get; set; }
    }
}

```

- `Property.cs`:

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class Property
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public decimal Price { get; set; }
        public int Bedrooms { get; set; }
    }
}
```

- `Favorite.cs`:

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class Favorite
    {
        public int Id { get; set; }
        public int UserId { get; set; }
        public int PropertyInfo { get; set; }
    }
}
```

- `PropertyImage.cs`:

```
csharpCopy code
namespace NexHomeAgent.Models
{
```

```
public class PropertyImage
{
    public int Id { get; set; }
    public int PropertyId { get; set; }
    public string ImageUrl { get; set; }
}
```

- `PropertyHistory.cs`:

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class PropertyHistory
    {
        public int Id { get; set; }
        public int PropertyId { get; set; }
        public string Description { get; set; }
        public DateTime Date { get; set; }
    }
}
```

## Step 2: Configure the Database Context in the API Project

### 1. Configure the Connection String:

- Add the connection string to `appsettings.json` in the `Dwellingly.API` project:

```
jsonCopy code
{
    "ConnectionStrings": {
        "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=DwellinglyDB;Trusted_Connection=True;MultipleActiveResultSets=true"
    }
}
```

```
        },
        "Logging": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft.AspNetCore": "Warning"
            }
        },
        "AllowedHosts": "*"
    }
}
```

## 2. Configure the Database Context in `Program.cs`:

- Update `Program.cs` in the `Dwellingly.API` project to register the `DbContext`:

```
csharpCopy code
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Serilog;
using NexHomeAgent.Data;

var builder = WebApplication.CreateBuilder(args);

// Configure Serilog
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Information()
    .WriteTo.Console()
    .WriteTo.File("logs/nexhomeagent.txt", rollingInterval: RollingInterval.Day)
    .CreateLogger();
builder.Host.UseSerilog();

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddDbContext<NexHomeAgentDbContext>(op
```

```

tions =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddApplicationInsightsTelemetry(options =>
{
    options.InstrumentationKey = builder.Configuration["ApplicationInsights:InstrumentationKey"];
});

var app = builder.Build();

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseHttpsRedirection();
app.UseRouting();
app.UseAuthorization();
app.UseAuthentication();

app.MapControllers();

app.Run();

```

## Step 3: Add Migrations and Update the Database

### 1. Add Entity Framework Tools:

- Install the `Microsoft.EntityFrameworkCore.Tools` package in the `Dwellingly.API` project via the NuGet Package Manager Console:

```
shCopy code  
Install-Package Microsoft.EntityFrameworkCore.Tools
```

## 2. Add Initial Migration:

- Open the Package Manager Console, select the `Dwellingly.API` project as the Default Project, and run:

```
shCopy code  
Add-Migration InitialCreate
```

## 3. Update the Database:

- Run the following command in the Package Manager Console to apply the migration and create the database:

```
shCopy code  
Update-Database
```

# Step 4: Seed Initial Data (Optional)

## 1. Create Seed Data Class:

- Add a new class `DbInitializer.cs` to the `Dwellingly.API` project to seed initial data:

```
csharpCopy code  
using Microsoft.Extensions.DependencyInjection;  
using NexHomeAgent.Models;  
  
public static class DbInitializer  
{  
    public static void Initialize(NexHomeAgentDbContext  
context)  
    {
```

```

        context.Database.EnsureCreated();

        // Check if users already exist
        if (context.Users.Any())
        {
            return;    // DB has been seeded
        }

        var users = new User[]
        {
            new User{Email="user1@example.com", Password="Password1"},
            new User{Email="user2@example.com", Password="Password2"},
        };

        foreach (var u in users)
        {
            context.Users.Add(u);
        }

        context.SaveChanges();

        var properties = new Property[]
        {
            new Property{Name="Property1", Address="Address1", Price=100000, Bedrooms=3},
            new Property{Name="Property2", Address="Address2", Price=200000, Bedrooms=4},
        };

        foreach (var p in properties)
        {
            context.Properties.Add(p);
        }
    }
}

```

```
        context.SaveChanges();
    }
}
```

## 2. Update `Program.cs` to Seed Data:

- Update `Program.cs` to call the seed method during application startup:

```
csharpCopy code
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Serilog;
using NexHomeAgent.Data;

var builder = WebApplication.CreateBuilder(args);

// Configure Serilog
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Information()
    .WriteTo.Console()
    .WriteTo.File("logs/nexhomeagent.txt", rollingInterval: RollingInterval.Day)
    .CreateLogger();
builder.Host.UseSerilog();

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddDbContext<NexHomeAgentDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddApplicationInsightsTelemetry(options =>
```

```

{
    options.InstrumentationKey = builder.Configuration
    ["ApplicationInsights:InstrumentationKey"];
});

var app = builder.Build();

// Seed the database
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<NexHomeAgentDbContext>();
    DbInitializer.Initialize(context);
}

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseHttpsRedirection();
app.UseRouting();
app.UseAuthorization();
app.UseAuthentication();

app.MapControllers();

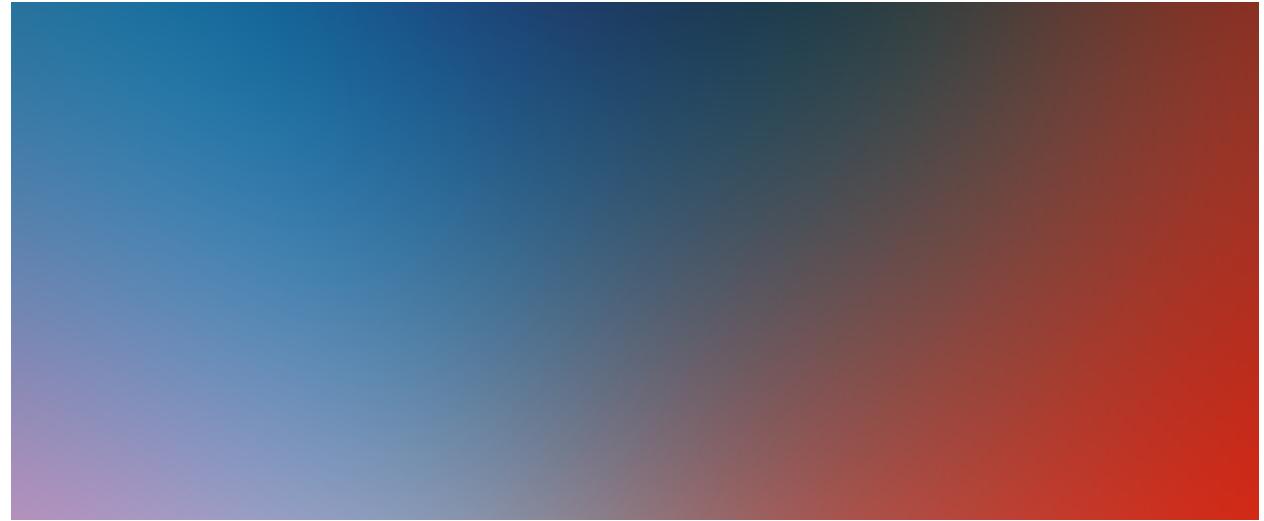
app.Run();

```

## Conclusion

Following these steps, you will set up the databases for the Dwellingly | AI application, including configuring Entity Framework Core, setting up migrations, and initializing the databases with

seed data. This ensures that your application has a robust and scalable database structure, ready for development and testing.



## **Phase 2: Personalized Property Recommendation System**



# MVP Development Plan for Dwellingly | AI

## Overview

To develop the MVP for Dwellingly | AI using the Microsoft technology stack within a 1-month timeframe, we will leverage various Microsoft tools and services to streamline development. Here's a detailed plan:

## Technology Stack

- **Backend:** ASP.NET Core
- **Frontend:** Blazor
- **Database:** Azure SQL Database
- **AI/ML:** Azure Machine Learning and Azure OpenAI (GPT-4)
- **Hosting and Infrastructure:** Azure App Service
- **DevOps:** Azure DevOps

## Timeline and Key Activities

### Week 1: Project Kickoff and Initial Setup

#### 1. Project Kickoff:

- Conduct a project kickoff meeting, assign roles and responsibilities, and set up project management tools and communication channels.

#### 2. Requirement Gathering:

- Conduct stakeholder interviews to gather core requirements.
- Document essential functional requirements.
- Create basic user stories and use cases.

### **3. System Architecture:**

- Design high-level system architecture using Microsoft technologies.
- Define key system components.
- Create initial wireframes for the user interface.

## **Week 2: Development Setup and Backend Implementation**

### **1. Development Environment Setup:**

- Set up development environments for the backend and frontend using Visual Studio.
- Configure version control systems with Azure Repos.
- Install necessary development tools and libraries.

### **2. Backend Development:**

- Implement core backend functionalities using ASP.NET Core.
- Develop data models and database schema in Azure SQL Database.
- Create API endpoints for property search and listings using canned data.
- Implement AI-driven valuation tools using Azure Machine Learning with simulated data.

## **Week 3: Frontend Implementation and Integration**

### **1. Frontend Development:**

- Implement core frontend components using Blazor.
- Develop user interface for property search and listings.
- Use mock data to simulate real estate properties.

### **2. Integration:**

- Integrate frontend with backend APIs using canned data.
- Ensure data flow and functionality using simulated data.
- Deploy backend and frontend to Azure App Service for hosting.

## **Week 4: Finalization and Demo Preparation**

### **1. Testing and Bug Fixing:**

- Conduct basic functionality tests to ensure the MVP works as intended.

- Fix any critical issues identified during testing.

## 2. Demo Preparation:

- Prepare demo scripts and presentations.
- Create marketing materials and documentation for the demo.
- Conduct internal demo to ensure readiness.

## 3. Final Adjustments:

- Make any final adjustments based on internal feedback.
- Ensure the MVP is polished and ready for presentation.

## 4. Investor Presentation:

- Present the sandboxed MVP to potential investors.
- Collect feedback and address any questions.
- Plan next steps based on investor feedback and interest.

# Detailed Instructions for Development

## Prerequisites

### Software

- Visual Studio 2022
- .NET 8 SDK
- Azure CLI
- Postman
- Git

### Accounts

- Azure Subscription
- GitHub or Azure Repos Account

## Development Environment Setup

### Step 1: Install Visual Studio 2022

1. Download and install Visual Studio 2022 from the [official website](#).
2. During installation, select the following workloads:
  - **ASP.NET and web development**
  - **Azure development**
  - **Data storage and processing**

## Step 2: Configure Azure CLI

1. Download and install the Azure CLI from the [official Azure documentation](#).
2. Sign in to your Azure account:

```
shCopy code  
az login
```

## Step 3: Set Up Git

1. Download and install Git from the [official Git website](#).
2. Configure Git with your user information:

```
shCopy code  
git config --global user.name "Your Name"  
git config --global user.email "your.email@example.com"
```

## Backend Development

### Step 1: Create ASP.NET Core Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **ASP.NET Core Web API**.
  - Choose **.NET 8.0** as the target framework.
  - Name the project `Dwellingly.API`.

## Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
shCopy code
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Dapper
Install-Package AutoMapper
Install-Package Microsoft.Data.SqlClient
```

## Step 3: Configure Entity Framework Core

1. Add a DbContext class:

```
csharpCopy code
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {

    }

    public DbSet<Property> Properties { get; set; }
}
```

1. Configure the connection string in `appsettings.json`:

```
jsonCopy code
"ConnectionStrings": {
    "DefaultConnection": "Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security Info=False;User ID=<your_user>;Password=<your_password>;MultipleActiveResultSets=True"
```

```
Sets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"  
}
```

1. Register the DbContext in [Program.cs](#):

```
csharpCopy code  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));  
    services.AddControllers();  
}
```

## Step 4: Create API Endpoints

1. Create a controller for property listings:

```
csharpCopy code  
[ApiController]  
[Route("api/[controller]")]  
public class PropertiesController : ControllerBase  
{  
    private readonly ApplicationContext _context;  
  
    public PropertiesController(ApplicationContext context)  
    {  
        _context = context;  
    }  
  
    [HttpGet]  
    public async Task<ActionResult<IEnumerable<Property>>> GetProperties()  
    {
```

```
        return await _context.Properties.ToListAsync();
    }
}
```

## Frontend Development

### Step 1: Create Blazor Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **Blazor WebAssembly App**.
  - Name the project `Dwellingly.Client`.

### Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
shCopy code
Install-Package Blazorise
Install-Package ChartJs.Blazor
Install-Package MudBlazor
```

### Step 3: Configure Blazor Components

1. Configure Blazorise in `Program.cs`:

```
csharpCopy code
builder.Services.AddBlazorise(options =>
{
    options.ChangeTextOnKeyPress = true;
})
.AddBootstrapProviders()
```

```
.AddFontAwesomeIcons();
```

## Step 4: Create Property Listing Component

1. Create a new Razor component `PropertyList.razor`:

```
razorCopy code
@page "/properties"
@inject HttpClient Http

<h3>Property Listings</h3>
<ul>
    @foreach (var property in properties)
    {
        <li>@property.Name - @property.Price</li>
    }
</ul>

@code {
    private List<Property> properties;

    protected override async Task OnInitializedAsync()
    {
        properties = await Http.GetFromJsonAsync<List<Property>>(
            "api/properties");
    }
}
```

## AI/ML Integration

### Step 1: Set Up Azure Machine Learning

1. Create an Azure Machine Learning workspace in the Azure portal.
2. Install the AzureML SDK for Python:

```
shCopy code  
pip install azureml-sdk
```

## Step 2: Train and Deploy Models

1. Use the AzureML SDK to train and deploy models:

```
pythonCopy code  
from azureml.core import Workspace, Experiment  
from azureml.train.automl import AutoMLConfig  
  
ws = Workspace.from_config()  
experiment = Experiment(ws, "property-valuation")  
  
automl_config = AutoMLConfig(  
    task="regression",  
    training_data=train_data,  
    label_column_name="price",  
    primary_metric="r2_score",  
    compute_target=compute_target,  
    max_trials=5  
)  
  
run = experiment.submit(automl_config, show_output=True)
```

## Hosting and Deployment

### Step 1: Deploy to Azure App Service

1. Right-click the project in Visual Studio and select **Publish**.
2. Choose **Azure** as the target.
3. Follow the prompts to create a new Azure App Service instance and deploy the application.

### Step 2: Configure CI/CD with Azure DevOps

1. Create a new project in Azure DevOps.
2. Set up a pipeline using the YAML file:

```
yamlCopy code
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '8.x'
    installationPath: $(Agent.ToolsDirectory)/dotnet

- script: dotnet build --configuration Release
  displayName: 'Build project'

- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

## References

For additional information and resources on developing ASP.NET Core and Blazor applications with Azure integration, refer to:

- [Blazor | Build client web apps with C#](#)
- [ASP.NET documentation | Microsoft Learn](#)
- [ASP.NET Core | Open-source web framework for .NET](#)
- [Error after deploying asp.net core app to azure](#)

This detailed developer guide ensures that the engineering team has all the necessary instructions and resources to successfully develop the Dwellingly | AI Sandboxed MVP using the specified Microsoft technologies.

## Detailed Instructions for Development

### Prerequisites

#### Software

- **Visual Studio 2022**
- **.NET 8 SDK**
- **Azure CLI**
- **Postman**
- **Git**

#### Accounts

- **Azure Subscription**
- **GitHub or Azure Repos Account**

## Development Environment Setup

### Step 1: Install Visual Studio 2022

1. Download and install Visual Studio 2022 from the [official website](#).
2. During installation, select the following workloads:
  - **ASP.NET and web development**
  - **Azure development**
  - **Data storage and processing**

### Step 2: Configure Azure CLI

1. Download and install the Azure CLI from the [official Azure documentation](#).
2. Sign in to your Azure account:

```
shCopy code  
az login
```

## Step 3: Set Up Git

1. Download and install Git from the [official Git website](#).
2. Configure Git with your user information:

```
shCopy code  
git config --global user.name "Your Name"  
git config --global user.email "your.email@example.com"
```

# Backend Development

## Step 1: Create ASP.NET Core Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **ASP.NET Core Web API**.
  - Choose **.NET 8.0** as the target framework.
  - Name the project **Dwellingly.API**.

## Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
shCopy code  
Install-Package Microsoft.EntityFrameworkCore  
Install-Package Microsoft.EntityFrameworkCore.SqlServer  
Install-Package Dapper  
Install-Package AutoMapper
```

```
Install-Package Microsoft.Data.SqlClient
```

## Step 3: Configure Entity Framework Core

1. Add a DbContext class:

```
csharpCopy code
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {

    }

    public DbSet<Property> Properties { get; set; }
}
```

1. Configure the connection string in `appsettings.json`:

```
jsonCopy code
"ConnectionStrings": {
    "DefaultConnection": "Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security Info=False;User ID=<your_user>;Password=<your_password>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
}
```

1. Register the DbContext in `Program.cs`:

```
csharpCopy code
public void ConfigureServices(IServiceCollection services)
{
```

```
        services.AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
        services.AddControllers();
    }
```

## Step 4: Create API Endpoints

1. Create a controller for property listings:

```
csharpCopy code
[ApiController]
[Route("api/[controller]")]
public class PropertiesController : ControllerBase
{
    private readonly ApplicationContext _context;

    public PropertiesController(ApplicationContext context)
    {
        _context = context;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<Property>>> GetProperties()
    {
        return await _context.Properties.ToListAsync();
    }
}
```

## Frontend Development

### Step 1: Create Blazor Project

1. Open Visual Studio 2022.

2. Create a new project:

- Select **Blazor WebAssembly App**.
- Name the project `Dwellingly.Client`.

## Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.

2. Install the following packages:

```
shCopy code
Install-Package Blazorise
Install-Package ChartJs.Blazor
Install-Package MudBlazor
```

## Step 3: Configure Blazor Components

1. Configure Blazorise in `Program.cs`:

```
csharpCopy code
builder.Services.AddBlazorise(options =>
{
    options.ChangeTextOnKeyPress = true;
})
.AddBootstrapProviders()
.AddFontAwesomeIcons();
```

## Step 4: Create Property Listing Component

1. Create a new Razor component `PropertyList.razor`:

```
razorCopy code
@page "/properties"
@inject HttpClient Http

<h3>Property Listings</h3>
```

```

<ul>
    @foreach (var property in properties)
    {
        <li>@property.Name - @property.Price</li>
    }
</ul>

@code {
    private List<Property> properties;

    protected override async Task OnInitializedAsync()
    {
        properties = await Http.GetFromJsonAsync<List<Property>>(
            "api/properties");
    }
}

```

## AI/ML Integration

### Step 1: Set Up Azure Machine Learning

1. Create an Azure Machine Learning workspace in the Azure portal.
2. Install the AzureML SDK for Python:

```

shCopy code
pip install azureml-sdk

```

### Step 2: Train and Deploy Models

1. Use the AzureML SDK to train and deploy models:

```

pythonCopy code
from azureml.core import Workspace, Experiment
from azureml.train.automl import AutoMLConfig

```

```

ws = Workspace.from_config()
experiment = Experiment(ws, "property-valuation")

automl_config = AutoMLConfig(
    task="regression",
    training_data=train_data,
    label_column_name="price",
    primary_metric="r2_score",
    compute_target=compute_target,
    max_trials=5
)

run = experiment.submit(automl_config, show_output=True)

```

## Hosting and Deployment

### Step 1: Deploy to Azure App Service

1. Right-click the project in Visual Studio and select **Publish**.
2. Choose **Azure** as the target.
3. Follow the prompts to create a new Azure App Service instance and deploy the application.

### Step 2: Configure CI/CD with Azure DevOps

1. Create a new project in Azure DevOps.
2. Set up a pipeline using the YAML file:

```

yamlCopy code
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UseDotNet@2

```

```

inputs:
  packageType: 'sdk'
  version: '8.x'
  installationPath: $(Agent.ToolsDirectory)/dotnet

- script: dotnet build --configuration Release
  displayName: 'Build project'

- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'

```

## References

For additional information and resources on developing ASP.NET Core and Blazor applications with Azure integration, refer to:

- [Blazor | Build client web apps with C#](#)
- [ASP.NET documentation | Microsoft Learn](#)
- [ASP.NET Core | Open-source web framework for .NET](#)
- [Error after deploying asp.net core app to azure](#)

This detailed developer guide ensures that the engineering team has all the necessary instructions and resources to successfully develop the Dwellingly | AI Sandboxed MVP using the specified Microsoft technologies.

```
import os
```

## Define the content for each file

```
content_mvp_plan = """"
```

### MVP Development Plan for Dwellingly | AI

# Overview

To develop the MVP for Dwellingly | AI using the Microsoft technology stack within a 1-month timeframe, we will leverage various Microsoft tools and services to streamline development. Here's a detailed plan:

## Technology Stack

- **Backend:** ASP.NET Core
- **Frontend:** Blazor
- **Database:** Azure SQL Database
- **AI/ML:** Azure Machine Learning and Azure OpenAI (GPT-4)
- **Hosting and Infrastructure:** Azure App Service
- **DevOps:** Azure DevOps

## Timeline and Key Activities

### Week 1: Project Kickoff and Initial Setup

#### 1. Project Kickoff:

- Conduct a project kickoff meeting, assign roles and responsibilities, and set up project management tools and communication channels.

#### 2. Requirement Gathering:

- Conduct stakeholder interviews to gather core requirements.
- Document essential functional requirements.
- Create basic user stories and use cases.

#### 3. System Architecture:

- Design high-level system architecture using Microsoft technologies.
- Define key system components.
- Create initial wireframes for the user interface.

### Week 2: Development Setup and Backend Implementation

#### 1. Development Environment Setup:

- Set up development environments for the backend and frontend using Visual Studio.
- Configure version control systems with Azure Repos.
- Install necessary development tools and libraries.

## **2. Backend Development:**

- Implement core backend functionalities using ASP.NET Core.
- Develop data models and database schema in Azure SQL Database.
- Create API endpoints for property search and listings using canned data.
- Implement AI-driven valuation tools using Azure Machine Learning with simulated data.

# **Week 3: Frontend Implementation and Integration**

## **1. Frontend Development:**

- Implement core frontend components using Blazor.
- Develop user interface for property search and listings.
- Use mock data to simulate real estate properties.

## **2. Integration:**

- Integrate frontend with backend APIs using canned data.
- Ensure data flow and functionality using simulated data.
- Deploy backend and frontend to Azure App Service for hosting.

# **Week 4: Finalization and Demo Preparation**

## **1. Testing and Bug Fixing:**

- Conduct basic functionality tests to ensure the MVP works as intended.
- Fix any critical issues identified during testing.

## **2. Demo Preparation:**

- Prepare demo scripts and presentations.
- Create marketing materials and documentation for the demo.
- Conduct internal demo to ensure readiness.

## **3. Final Adjustments:**

- Make any final adjustments based on internal feedback.
- Ensure the MVP is polished and ready for presentation.

#### **4. Investor Presentation:**

- Present the sandboxed MVP to potential investors.
  - Collect feedback and address any questions.
  - Plan next steps based on investor feedback and interest.
- .....

content\_prerequisites = "'''

## **Detailed Instructions for Development**

### **Prerequisites**

#### **Software**

- **Visual Studio 2022**
- **.NET 8 SDK**
- **Azure CLI**
- **Postman**
- **Git**

#### **Accounts**

- **Azure Subscription**
  - **GitHub or Azure Repos Account**
- .....

content\_setup = "'''

## **Development Environment Setup**

### **Step 1: Install Visual Studio 2022**

1. Download and install Visual Studio 2022 from the [official website](#).
2. During installation, select the following workloads:

- **ASP.NET and web development**
- **Azure development**
- **Data storage and processing**

## Step 2: Configure Azure CLI

1. Download and install the Azure CLI from the [official Azure documentation](#).
2. Sign in to your Azure account:

```
az login
```

## Step 3: Set Up Git

1. Download and install Git from the official Git website.
2. Configure Git with your user information:

```
shAlways show details
Copy codegit config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

"""  
content\_backend = """"

## Backend Development

### Step 1: Create ASP.NET Core Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **ASP.NET Core Web API**.
  - Choose **.NET 8.0** as the target framework.
  - Name the project **Dwellingly.API**.

## Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
shAlways show details
Copy codeInstall-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Dapper
Install-Package AutoMapper
Install-Package Microsoft.Data.SqlClient
```

## Step 3: Configure Entity Framework Core

1. Add a DbContext class:

```
csharpAlways show details
Copy codepublic class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {

    }

    public DbSet<Property> Properties { get; set; }
}
```

1. Configure the connection string in `appsettings.json`:

```
jsonAlways show details
Copy code"ConnectionStrings": {
    "DefaultConnection": "Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security Info=False;User ID=<your_user>;Password=<your_password>;MultipleActiveResultSets=True"
```

```
Sets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"  
}
```

1. Register the DbContext in `Program.cs`:

```
csharpAlways show details  
Copy codepublic void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));  
    services.AddControllers();  
}
```

## Step 4: Create API Endpoints

1. Create a controller for property listings:

```
csharpAlways show details  
Copy code[ApiController]  
[Route("api/[controller]")]  
public class PropertiesController : ControllerBase  
{  
    private readonly ApplicationContext _context;  
  
    public PropertiesController(ApplicationContext context)  
    {  
        _context = context;  
    }  
  
    [HttpGet]  
    public async Task<ActionResult<IEnumerable<Property>>> GetProperties()
```

```
    {
        return await _context.Properties.ToListAsync();
    }
}

"""

content_frontend = """
```

## Frontend Development

### Step 1: Create Blazor Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **Blazor WebAssembly App**.
  - Name the project `Dwellingly.Client`.

### Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
shAlways show details
Copy codeInstall-Package Blazorise
Install-Package ChartJs.Blazor
Install-Package MudBlazor
```

### Step 3: Configure Blazor Components

1. Configure Blazorise in `Program.cs`:

```
csharpAlways show details
Copy codebuilder.Services.AddBlazorise(options =>
{
    options.ChangeTextOnKeyPress = true;
```

```
    })
    .AddBootstrapProviders()
    .AddFontAwesomeIcons();
```

## Step 4: Create Property Listing Component

1. Create a new Razor

Always show details  
Copy code

# Define the content for each code file

content\_backend = "''''

## Backend Development

### Step 1: Create ASP.NET Core Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select ASP.NET Core Web API.
  - Choose **.NET 8.0** as the target framework.
  - Name the project `Dwellingly.API`.

### Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

sh

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Dapper
```

Install-Package AutoMapper

Install-Package Microsoft.Data.SqlClient

## Step 3: Configure Entity Framework Core

1. Add a DbContext class:

```
csharpAlways show details
Copy codepublic class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    public DbSet<Property> Properties { get; set; }
}
```

1. Configure the connection string in `appsettings.json`:

```
jsonAlways show details
Copy code"ConnectionStrings": {
    "DefaultConnection": "Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security Info=False;User ID=<your_user>;Password=<your_password>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
}
```

1. Register the DbContext in `Program.cs`:

```
csharpAlways show details
Copy codepublic void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("D
```

```
efaultConnection"")));
    services.AddControllers();
}
```

## Step 4: Create API Endpoints

1. Create a controller for property listings:

```
csharpAlways show details
Copy code[ApiController]
[Route("api/[controller]")]
public class PropertiesController : ControllerBase
{
    private readonly ApplicationDbContext _context;

    public PropertiesController(ApplicationDbContext context)
    {
        _context = context;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<Property>>> GetProperties()
    {
        return await _context.Properties.ToListAsync();
    }
}

.....
content_frontend = """
```

## Frontend Development

### Step 1: Create Blazor Project

1. Open Visual Studio 2022.

2. Create a new project:

- Select **Blazor WebAssembly App**.
- Name the project `Dwellingly.Client`.

## Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
shAlways show details
Copy codeInstall-Package Blazorise
Install-Package ChartJs.Blazor
Install-Package MudBlazor
```

## Step 3: Configure Blazor Components

1. Configure Blazorise in `Program.cs`:

```
csharpAlways show details
Copy codebuilder.Services.AddBlazorise(options =>
{
    options.ChangeTextOnKeyPress = true;
})
.AddBootstrapProviders()
.AddFontAwesomeIcons();
```

## Step 4: Create Property Listing Component

1. Create a new Razor

```
Always show details
Copy code
```

just the code file types

# Define the content for each code file

```
content_backend = """"
```

## Backend Development

### Step 1: Create ASP.NET Core Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **ASP.NET Core Web API**.
  - Choose **.NET 8.0** as the target framework.
  - Name the project **Dwellingly.API**.

### Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Dapper
Install-Package AutoMapper
Install-Package Microsoft.Data.SqlClient
```

### Step 3: Configure Entity Framework Core

1. Add a DbContext class:

```
csharpAlways show details
Copy codepublic class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

```
        public DbSet<Property> Properties { get; set; }  
    }
```

1. Configure the connection string in `appsettings.json`:

```
jsonAlways show details  
Copy code"ConnectionStrings": {  
    "DefaultConnection": "Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security Info=False;  
    User ID=<your_user>;Password=<your_password>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"  
}
```

1. Register the DbContext in `Program.cs`:

```
csharpAlways show details  
Copy codepublic void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));  
    services.AddControllers();  
}
```

## Step 4: Create API Endpoints

1. Create a controller for property listings:

```
csharpAlways show details  
Copy code[ApiController]  
[Route("api/[controller]")]
```

```

public class PropertiesController : ControllerBase
{
    private readonly ApplicationDbContext _context;

    public PropertiesController(ApplicationDbContext context)
    {
        _context = context;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<Property>>> GetProperties()
    {
        return await _context.Properties.ToListAsync();
    }
}

```

.....

content\_frontend = "''''

## Frontend Development

### Step 1: Create Blazor Project

1. Open Visual Studio 2022.
2. Create a new project:
  - Select **Blazor WebAssembly App**.
  - Name the project `Dwellingly.Client`.

### Step 2: Install Required NuGet Packages

1. Open the NuGet Package Manager Console.
2. Install the following packages:

```

shAlways show details
Copy codeInstall-Package Blazorise

```

```
Install-Package ChartJs.Blazor  
Install-Package MudBlazor
```

## Step 3: Configure Blazor Components

1. Configure Blazorise in `Program.cs`:

```
csharpAlways show details  
Copy codebuilder.Services.AddBlazorise(options =>  
{  
    options.ChangeTextOnKeyPress = true;  
})  
.AddBootstrapProviders()  
.AddFontAwesomeIcons();
```

## Step 4: Create Property Listing Component

1. Create a new Razor component `PropertyList.razor`:

```
razorAlways show details  
Copy code@page "/properties"  
@inject HttpClient Http  
  
<h3>Property Listings</h3>  
<ul>  
    @foreach (var property in properties)  
    {  
        <li>@property.Name - @property.Price</li>  
    }  
</ul>  
  
@code {  
    private List<Property> properties;  
  
    protected override async Task OnInitializedAsync()
```

```

    {
        properties = await Http.GetFromJsonAsync<List<Property>>
("api/properties");
    }
}

"""

content_ai_ml = """

```

## AI/ML Integration

### Step 1: Set Up Azure Machine Learning

1. Create an Azure Machine Learning workspace in the Azure portal.
2. Install the AzureML SDK for Python:

```

shAlways show details
Copy codepip install azureml-sdk

```

### Step 2: Train and Deploy Models

1. Use the AzureML SDK to train and deploy models:

```

pythonAlways show details
Copy codefrom azureml.core import Workspace, Experiment
from azureml.train.automl import AutoMLConfig

ws = Workspace.from_config()
experiment = Experiment(ws, "property-valuation")

automl_config = AutoMLConfig(
    task="regression",
    training_data=train_data,
    label_column_name="price",
    primary_metric="r2_score",
    compute_target=compute_target,

```

```

    max_trials=5
)

run = experiment.submit(automl_config, show_output=True)

"""

content_devops = """

```

## Hosting and Deployment

### Step 1: Deploy to Azure App Service

1. Right-click the project in Visual Studio and select **Publish**.
2. Choose **Azure** as the target.
3. Follow the prompts to create a new Azure App Service instance and deploy the application.

### Step 2: Configure CI/CD with Azure DevOps

1. Create a new project in Azure DevOps.
2. Set up a pipeline using the YAML file:

```

yamlAlways show details
Copy codetrigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '8.x'
    installationPath: $(Agent.ToolsDirectory)/dotnet

- script: dotnet build --configuration Release

```

```

displayName: 'Build project'

- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'

```

.....

## Save each content to a separate file

```

file_paths = {
  "backend_development.md": content_backend,
  "frontend_development.md": content_frontend,
  "ai_ml_integration.md": content_ai_ml,
  "devops.md": content_devops
}

for filename, content in file_paths.items():
  with open(f"/mnt/data/{filename}", "w") as file:
    file.write(content)

file_paths

```

[Always show details](#)

[Copy code](#)

### Result

```

{'backend_development.md': '\n### Backend Development\n\n#### Step 1: Create ASP.NET Core Project\n1. Open Visual Studio 2022.\n2. Create a new project:\n  - Select **ASP.NET Core Web API**.\n  - Choose **.NET 8.0** as the target framework.\n  - Name the project `Dwellingly.API`.\n\n#### Step 2: Install Required NuGet Packages\n1. Open the NuGet Package Manager Console.\n2. Install the following packages:\n  - `Microsoft.EntityFrameworkCore` (Install-Package Microsoft.EntityFrameworkCore)\n  - `Microsoft.EntityFrameworkCore.SqlServer` (Install-Package Microsoft.EntityFrameworkCore.SqlServer)\n  - `Microsoft.EntityFrameworkCore.Dapper` (Install-Package Microsoft.EntityFrameworkCore.Dapper)\n  - `AutoMapper` (Install-Package AutoMapper)\n  - `Microsoft.Data.SqlClient` (Install-Package Microsoft.Data.SqlClient)\n\n#### Step 3: Configure Entity Framework Core\n1. Add a `DbContext` class:\n```csharp\npublic class ApplicationDbContext : DbContext\n{\n  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)\n  {\n    public DbSet<Property> Properties { get; set; }\n  }\n}\n```\n2. Configure the connection string in `appsettings.json`:\n```json\n"ConnectionStrings": {\n  "DefaultConnection": ...

```

```

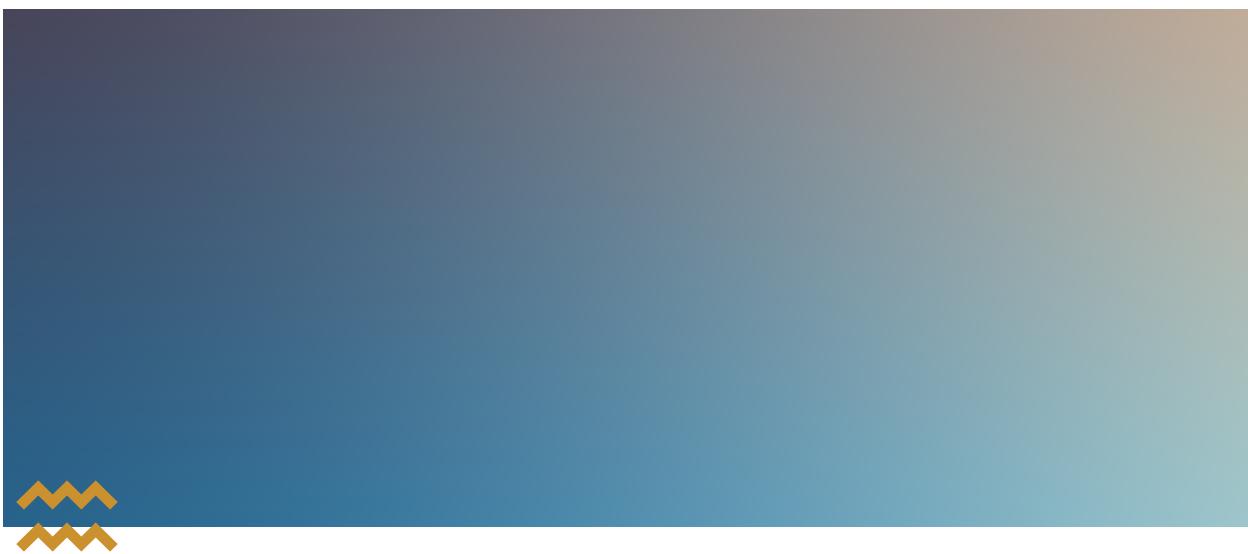
"Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security
Info=False;User ID=<your_user>;Password=<your_password>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection
Timeout=30;"\n}\n```\n\n3. Register the DbContext in `Program.cs`:\n```csharp\npublic void
ConfigureServices(IServiceCollection services)\n{\n    services.AddDbContext<ApplicationContext>
(options =>\n        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));\n    services.AddControllers();\n}\n```\n#### Step 4: Create API Endpoints\n1. Create a controller
for property listings:\n```csharp\n[ApiController]\n[Route("api/[controller]")]  
public class
PropertiesController : ControllerBase\n{\n    private readonly ApplicationContext _context;\n  
    public PropertiesController(ApplicationContext context)\n    {\n        _context = context;\n    }\n  
    [HttpGet]\n    public async Task<ActionResult<IEnumerable<Property>>> GetProperties()\n    {\n        return await _context.Properties.ToListAsync();\n    }\n}\n```\n'frontend_development.md': '\n### Frontend Development\n\n1. Create Blazor Project\n1. Open Visual Studio 2022.\n2. Create a new project:\n   - Select **Blazor WebAssembly App**.\n   - Name the project `Dwellingly.Client`.\n\n### Step 2: Install Required NuGet Packages\n1. Open the NuGet Package Manager Console.\n2. Install the following packages:\n```sh\nInstall-Package Blazorfse\nInstall-Package ChartJs.Blazor\nInstall-Package MudBlazor\n```\n### Step 3: Configure Blazor Components\n1. Configure Blazorfse in `Program.cs`:\n```csharp\nbuilder.Services.AddBlazorfse(options =>\n    options.ChangeTextOnKeyPress =
true;)\n    .AddBootstrapProviders()\n    .AddFontAwesomeIcons();\n```\n### Step 4: Create Property Listing Component\n1. Create a new Razor component `PropertyList.razor`:\n```razor\n@page "/properties"\n@inject HttpClient Http\n

### Property Listings



- @foreach (var
property in properties)\n    <li>@property.Name - @property.Price</li>\n</ul>\n@code {\n    private List<Property> properties;\n    protected override async Task
OnInitializedAsync()\n    {\n        properties = await Http.GetFromJsonAsync<List<Property>>(
"api/properties");\n    }\n}
\n```\n'ai_ml_integration.md': '\n### AI/ML Integration\n\n1. Set Up Azure Machine
Learning\n1. Create an Azure Machine Learning workspace in the Azure portal.\n2. Install the
AzureML SDK for Python:\n```sh\npip install azureml-sdk\n```\n### Step 2: Train and Deploy
Models\n1. Use the AzureML SDK to train and deploy models:\n```python\nfrom azureml.core import
Workspace, Experiment\nfrom azureml.train.automl import AutoMLConfig\nws =
Workspace.from_config()\nexperiment = Experiment(ws, "property-valuation")\nautoml_config =
AutoMLConfig(\n    task="regression",\n    training_data=train_data,\n    label_column_name="price",
primary_metric="r2_score",\n    compute_target=compute_target,\n    max_trials=5\n)\nrun = experiment.submit(automl_config, show_output=True)\n```\n'devops.md': '\n### Hosting and Deployment\n\n1. Deploy to

```



# Complete Backend Setup for Dwellingly | AI in Visual Studio

Below is the complete backend setup for the Dwellingly | AI application, which includes the project structure, configurations, services, middleware, and controllers.

## Step-by-Step Guide

### Project Structure

```
luaCopy code
Dwellingly
| -- Controllers/
|   | -- PropertyController.cs
| -- Data/
|   | -- NexHomeAgentContext.cs
| -- Middleware/
|   | -- ExceptionMiddleware.cs
|   | -- RequestLoggingMiddleware.cs
| -- Models/
|   | -- ChatMessage.cs
```

```
|   |-- Property.cs
|   |-- User.cs
|   |-- Favorite.cs
|   |-- PropertyImage.cs
|   |-- PropertyHistory.cs
|-- Services/
|   |-- ChatService.cs
|   |-- IChatService.cs
|   |-- PropertyCommandService.cs
|   |-- PropertyQueryService.cs
|-- Program.cs
|-- appsettings.json
```

## 1. Define the Database Context and Models

Data/NexHomeAgentContext.cs

```
csharpCopy code
using Microsoft.EntityFrameworkCore;
using NexHomeAgent.Models;

namespace NexHomeAgent.Data
{
    public class NexHomeAgentContext : DbContext
    {
        public NexHomeAgentContext(DbContextOptions<NexHomeAgentContext> options)
            : base(options)
        {

        }

        public DbSet<Property> Properties { get; set; }
        public DbSet<ChatMessage> ChatMessages { get; set; }
        public DbSet<User> Users { get; set; }
        public DbSet<Favorite> Favorites { get; set; }
```

```
        public DbSet<PropertyImage> PropertyImages { get; set; }
        public DbSet<PropertyHistory> PropertyHistories { get; set; }
    }
```

### Models/Property.cs

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class Property
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Location { get; set; }
        public int Bedrooms { get; set; }
        public int Bathrooms { get; set; }
        public string PhotoUrl { get; set; }
    }
}
```

### Models/ChatMessage.cs

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class ChatMessage
    {
        public int Id { get; set; }
        public string Text { get; set; }
    }
}
```

```
        public bool IsUser { get; set; }
        public DateTime Timestamp { get; set; }
    }
}
```

### Models/User.cs

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class User
    {
        public int Id { get; set; }
        public string Email { get; set; }
        public string Password { get; set; }
    }
}
```

### Models/Favorite.cs

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class Favorite
    {
        public int Id { get; set; }
        public int UserId { get; set; }
        public int PropertyId { get; set; }
    }
}
```

### Models/PropertyImage.cs

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class PropertyImage
    {
        public int Id { get; set; }
        public int PropertyId { get; set; }
        public string ImageUrl { get; set; }
    }
}
```

### Models/PropertyHistory.cs

```
csharpCopy code
namespace NexHomeAgent.Models
{
    public class PropertyHistory
    {
        public int Id { get; set; }
        public int PropertyId { get; set; }
        public string Description { get; set; }
        public DateTime Date { get; set; }
    }
}
```

## 2. Configure Entity Framework Core

### 1. Install Entity Framework Core NuGet Packages:

- Open the **NuGet Package Manager Console** (Tools > NuGet Package Manager > Package Manager Console).
- Run the following commands:

```
shCopy code
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools
```

## 2. Add Connection String to `appsettings.json`:

- Add the connection string to `appsettings.json`:

```
jsonCopy code
{
  "ConnectionStrings": {
    "AzureSqlDatabase": "Server=your_server_name;Database=DwellinglyDB;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "OpenAI": {
    "ApiKey": "YOUR_API_KEY"
  }
}
```

## 3. Configure `Program.cs`:

- Update `Program.cs` to register the `DbContext` and configure other services:

```
csharpCopy code
using Microsoft.AspNetCore.Builder;
```

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Serilog;
using Azure.Identity;
using Azure.Security.KeyVault.Secrets;
using NexHomeAgent.Middleware;
using NexHomeAgent.Data;
using NexHomeAgent.Services;

var builder = WebApplication.CreateBuilder(args);

// Configure Serilog
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Information()
    .WriteTo.Console()
    .WriteTo.File("logs/nexhomeagent.txt", rollingInterval: RollingInterval.Day)
    .CreateLogger();
builder.Host.UseSerilog();

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddAuthentication(AzureADB2CDefaults.BearerAuthenticationScheme)
    .AddAzureADB2CBearer(options =>
{
    options.Instance = builder.Configuration["AzureAdB2C:Instance"];
    options.ClientId = builder.Configuration["AzureAdB2C:ClientId"];
    options.Domain = builder.Configuration["AzureAdB2C:Domain"];
    options.SignUpSignInPolicyId = builder.Configuration["AzureAdB2C:SignUpSignInPolicyId"];
});
```

```

builder.Services.AddDbContext<NexHomeAgentContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("AzureSqlDatabase")));

builder.Services.AddHttpClient<IChatService, ChatService>(client =>
{
    client.BaseAddress = new Uri("https://api.openai.com/v1/");
    client.DefaultRequestHeaders.Add("Authorization",
    "Bearer " + builder.Configuration["OpenAI:ApiKey"]);
});

builder.Services.AddApplicationInsightsTelemetry(options =>
{
    options.InstrumentationKey = builder.Configuration["ApplicationInsights:InstrumentationKey"];
});

var app = builder.Build();

// Seed the database
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<NexHomeAgentContext>();
    DbInitializer.Initialize(context);
}

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

```

```
}

app.UseHttpsRedirection();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.UseMiddleware<ExceptionMiddleware>();

app.MapControllers();

app.Run();
```

#### 4. Add Migrations and Update the Database:

- Open the **Package Manager Console** and run:

```
shCopy code
Add-Migration InitialCreate
Update-Database
```

### 3. Seed Initial Data

#### 1. Create Seed Data Class:

- Add a new class `DbInitializer.cs` in the `Data` folder:

```
csharpCopy code
using Microsoft.Extensions.DependencyInjection;
using NexHomeAgent.Models;

namespace NexHomeAgent.Data
{
    public static class DbInitializer
    {
```

```

        public static void Initialize(NexHomeAgentContext context)
        {
            context.Database.EnsureCreated();

            // Check if users already exist
            if (context.Users.Any())
            {
                return;    // DB has been seeded
            }

            var users = new User[]
            {
                new User{Email="user1@example.com", Password="Password1"},
                new User{Email="user2@example.com", Password="Password2"},
            };

            foreach (var u in users)
            {
                context.Users.Add(u);
            }

            context.SaveChanges();

            var properties = new Property[]
            {
                new Property{Title="Property1", Description="Description1", Price=100000, Location="Location 1", Bedrooms=3, Bathrooms=2, PhotoUrl="URL1"},
                new Property{Title="Property2", Description="Description2", Price=200000, Location="Location 2", Bedrooms=4, Bathrooms=3, PhotoUrl="URL2"},
            };
        }
    }
}

```

```

        foreach (var p in properties)
        {
            context.Properties.Add(p);
        }

        context.SaveChanges();
    }
}

```

## 2. Update `Program.cs` to Seed Data:

- Ensure the database seeding is called during application startup:

```

csharpCopy code
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Serilog;
using Azure.Identity;
using Azure.Security.KeyVault.Secrets;
using NexHomeAgent.Middleware;
using NexHomeAgent.Data;
using NexHomeAgent.Services;

var builder = WebApplication.CreateBuilder(args);

// Configure Serilog
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Information()
    .WriteTo.Console()
    .WriteTo.File("logs/nexhomeagent.txt", rollingInterval: RollingInterval.Day)
    .CreateLogger();
builder.Host.UseSerilog();

```

```

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddAuthentication(AzureADB2CDefaults.BearerAuthenticationScheme)
    .AddAzureADB2CBearer(options =>
{
    options.Instance = builder.Configuration["AzureAdB2C:Instance"];
    options.ClientId = builder.Configuration["AzureAdB2C:ClientId"];
    options.Domain = builder.Configuration["AzureAdB2C:Domain"];
    options.SignUpSignInPolicyId = builder.Configuration["AzureAdB2C:SignUpSignInPolicyId"];
});

builder.Services.AddDbContext<NexHomeAgentContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("AzureSqlDatabase")));
}

builder.Services.AddHttpClient<IChatService, ChatService>(client =>
{
    client.BaseAddress = new Uri("https://api.openai.com/v1/");
    client.DefaultRequestHeaders.Add("Authorization", "Bearer " + builder.Configuration["OpenAI:ApiKey"]);
});

builder.Services.AddApplicationInsightsTelemetry(options =>
{
    options.InstrumentationKey = builder.Configuration["ApplicationInsights:InstrumentationKey"];
}

```

```
});

var app = builder.Build();

// Seed the database
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<NexHomeAgentContext>();
    DbInitializer.Initialize(context);
}

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseHttpsRedirection();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.UseMiddleware<ExceptionMiddleware>();

app.MapControllers();

app.Run();
```

## 4. Implement Middleware

Middleware/ExceptionMiddleware.cs

```
csharpCopy code
using Microsoft.AspNetCore.Http;
using Serilog;
using System;
using System.Threading.Tasks;

namespace NexHomeAgent.Middleware
{
    public class ExceptionMiddleware
    {
        private readonly RequestDelegate _next;

        public ExceptionMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            try
            {
                await _next(context);
            }
            catch (Exception ex)
            {
                Log.Error(ex, "An unhandled exception occurred.");
                context.Response.StatusCode = StatusCodes.Status500InternalServerError;
                await context.Response.WriteAsync("An error occurred. Please try again later.");
            }
        }
    }
}
```

```
}
```

## 5. Implement Controllers and Services

### Controllers/PropertyController.cs

```
csharpCopy code
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using NexHomeAgent.Models;
using NexHomeAgent.Services;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace NexHomeAgent.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class PropertyController : ControllerBase
    {
        private readonly IPropertyCommandService _propertyCommandService;
        private readonly IPropertyQueryService _propertyQueryService;
        private readonly ILogger<PropertyController> _logger;

        public PropertyController(
            IPropertyCommandService propertyCommandService,
            IPropertyQueryService propertyQueryService,
            ILogger<PropertyController> logger)
        {
            _propertyCommandService = propertyCommandService;
            _propertyQueryService = propertyQueryService;
            _logger = logger;
        }
    }
}
```

```

        [HttpGet("search")]
        public async Task<ActionResult<IEnumerable<Property>>>
> Search([FromQuery] PropertySearchQuery query)
{
    _logger.LogInformation("Searching for properties
with criteria: {criteria}", query);
    var properties = await _propertyQueryService.Sear
chPropertiesAsync(query);
    return Ok(properties);
}

        [HttpGet("{id}")]
        public async Task<IActionResult> GetProperty(int id)
{
    _logger.LogInformation("Fetching property with I
D: {id}", id);
    var property = await _propertyQueryService.GetPropertyAsync(id);
    if (property == null)
    {
        _logger.LogWarning("Property with ID {id} not
found", id);
        return NotFound();
    }
    return Ok(property);
}

        [HttpPost]
        public async Task<IActionResult> CreateProperty(Create
PropertyCommand command)
{
    _logger.LogInformation("Creating a new property:
{property}", command);
    await _propertyCommandService.CreatePropertyAsync
(command);
}

```

```
        return CreatedAtAction(nameofGetProperty), new {
            id = command.Id }, command);
    }
}
}
```

### Services/PropertyCommandService.cs

```
csharpCopy code
using NexHomeAgent.Models;
using NexHomeAgent.Data;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace NexHomeAgent.Services
{
    public class PropertyCommandService : IPropertyCommandService
    {
        private readonly NexHomeAgentContext _context;
        private readonly ILogger<PropertyCommandService> _logger;

        public PropertyCommandService(NexHomeAgentContext context, ILogger<PropertyCommandService> logger)
        {
            _context = context;
            _logger = logger;
        }

        public async Task CreatePropertyAsync(CreatePropertyCommand command)
        {
            _logger.LogInformation("Creating a new property: {property}", command);
```

```

        var property = new Property
        {
            Title = command.Title,
            Description = command.Description,
            Price = command.Price,
            Location = command.Location,
            Bedrooms = command.Bedrooms,
            Bathrooms = command.Bathrooms,
            PhotoUrl = command.PhotoUrl
        };
        _context.Properties.Add(property);
        await _context.SaveChangesAsync();
    }
}

```

### Services/PropertyQueryService.cs

```

csharpCopy code
using NexHomeAgent.Models;
using NexHomeAgent.Data;
using Microsoft.Extensions.Logging;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace NexHomeAgent.Services
{
    public class PropertyQueryService : IPropertyQueryService
    {
        private readonly NexHomeAgentContext _context;
        private readonly ILogger<PropertyQueryService> _logger;
        public PropertyQueryService(NexHomeAgentContext conte

```

```

        xt, ILogger<PropertyQueryService> logger)
    {
        _context = context;
        _logger = logger;
    }

    public async Task<IEnumerable<Property>> SearchPropertiesAsync(PropertySearchQuery query)
    {
        _logger.LogInformation("Searching for properties
with criteria: {criteria}", query);
        return await _context.Properties
            .Where(p => p.Location.Contains(query.Location) &&
                p.Price >= query.MinPrice &&
                p.Price <= query.MaxPrice &&
                p.Bedrooms == query.Bedrooms)
            .ToListAsync();
    }

    public async Task<Property> GetPropertyAsync(int id)
    {
        _logger.LogInformation("Fetching property with ID: {id}", id);
        return await _context.Properties.FindAsync(id);
    }
}

```

## 6. Interface Definitions

### **Services/IPropertyCommandService.cs**

```

csharpCopy code
using NexHomeAgent.Models;

```

```
using System.Threading.Tasks;

namespace NexHomeAgent.Services
{
    public interface IPropertyCommandService
    {
        Task CreatePropertyAsync(CreatePropertyCommand command);
    }
}
```

### Services/IPropertyQueryService.cs

```
csharpCopy code
using NexHomeAgent.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace NexHomeAgent.Services
{
    public interface IPropertyQueryService
    {
        Task<IEnumerable<Property>> SearchPropertiesAsync(PropertySearchQuery query);
        Task<Property> GetPropertyAsync(int id);
    }
}
```

## Conclusion

This guide provides a complete setup for the backend of the Dwellingly | AI application using Visual Studio. It includes the configuration for Entity Framework Core, the necessary models, services, middleware, and controllers. This setup ensures a scalable, secure, and maintainable backend for your application.

Please let me know if you need further assistance or additional features!

This guide provides detailed instructions to set up the backend for the Dwellingly | AI application in Visual Studio, including creating the project, configuring Entity Framework Core, and integrating services.



# Set up the front end w/ requirements in Visual Studio

## Step 1: Create a Blazor Project

1. Open Visual Studio.
2. Create a new Blazor WebAssembly App project.
3. Name your project and configure the solution.

## Step 2: Add npm Support

1. Right-click on the project in the Solution Explorer.
2. Select "Add" > "New Item".
3. Choose "npm Configuration File" and name it `package.json`.

## Step 3: Configure `package.json`

Add the required dependencies to your `package.json` file. For example:

```
{  
  "name": "BlazorApp",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "build:css": "npx tailwindcss build src/tailwind.css -o wwwroot/css/tailwind.css",  
    "build": "npm run build:css"  
  },  
  "dependencies": {  
    "bootstrap": "^5.1.0",  
    "tailwindcss": "^2.2.19"  
  },  
  "devDependencies": {  
    "terser-webpack-plugin": "^5.1.4",  
    "css-minimizer-webpack-plugin": "^3.0.2"  
  }  
}
```

## Step 4: Install npm Dependencies

Create a script to install npm dependencies. You can use PowerShell or a batch script. Below is an example of a PowerShell script:

Create a file named `install-dependencies.ps1`:

```
# PowerShell script to install npm dependencies  
Write-Output "Installing npm dependencies..."  
  
# Navigate to the project directory  
Set-Location -Path $PSScriptRoot  
  
# Run npm install  
npm install
```

```
Write-Output "Dependencies installed successfully."
```

## Step 5: Configure Tailwind CSS

Create a Tailwind CSS configuration file by running the following command in the terminal:

```
npx tailwindcss init
```

This will create a `tailwind.config.js` file. Configure it as needed:

```
module.exports = {  
  purge: ['./**/*.razor', './**/*.html'],  
  darkMode: false, // or 'media' or 'class'  
  theme: {  
    extend: {},  
  },  
  variants: {  
    extend: {},  
  },  
  plugins: [],  
};
```

Create a `src/tailwind.css` file and add the Tailwind directives:

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;
```

## Step 6: Add Tailwind and Bootstrap to Your Project

Reference the compiled CSS files in your `index.html` or `_Host.cshtml`:

```
<!DOCTYPE html>  
<html>  
<head>
```

```

<link href="css/bootstrap.min.css" rel="stylesheet" />
<link href="css/tailwind.css" rel="stylesheet" />
<!-- Your other head content -->
</head>
<body>
    <app>Loading...</app>
    <script src="_framework/blazor.webassembly.js"></script>
</body>
</html>

```

## Step 7: Automate CSS Build Process

Ensure your `build:css` script runs when the project builds. You can modify the `.csproj` file to include npm commands:

Edit your `.csproj` file to include the npm script execution:

```

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

    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <OutputType>Exe</OutputType>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly" Version="6.0.0" />
        <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer" Version="6.0.0" PrivateAssets="all" />
    </ItemGroup>

    <Target Name="NpmInstall" BeforeTargets="Build">
        <Exec Command="npm install" />
    
```

```

</Target>

<Target Name="TailwindBuild" AfterTargets="Build">
  <Exec Command="npm run build:css" />
</Target>

</Project>

```

## Step 8: Run the Script and Build the Project

1. Run the PowerShell script `install-dependencies.ps1` to install npm dependencies:

```
.\\install-dependencies.ps1
```

1. Build your project in Visual Studio. The `NpmInstall` and `TailwindBuild` targets will ensure that npm dependencies are installed and Tailwind CSS is built before the Blazor app is built.

## Summary

By following these steps, you have set up a Blazor project in Visual Studio with npm support, added Bootstrap and Tailwind CSS, and automated the build process for your CSS files. This ensures that your front-end dependencies are correctly installed and your styles are up-to-date whenever you build your project.

## 1. Property Search Component (`PropertySearch.razor`)

```

@page "/property-search"
@inject IPropertyService PropertyService

<h2>Property Search</h2>

<EditForm Model="@searchCriteria" OnValidSubmit="@SearchPropertiesAsync">
  <DataAnnotationsValidator />
  <ValidationSummary />

```

```

<div class="form-group">
    <label for="location">Location:</label>
    <InputText id="location" class="form-control" @bind-Value="searchCriteria.Location" />
</div>
<div class="form-group">
    <label for="minPrice">Min Price:</label>
    <InputNumber id="minPrice" class="form-control" @bind-Value="searchCriteria.MinPrice" />
</div>
<div class="form-group">
    <label for="maxPrice">Max Price:</label>
    <InputNumber id="maxPrice" class="form-control" @bind-Value="searchCriteria.MaxPrice" />
</div>
<div class="form-group">
    <label for="bedrooms">Bedrooms:</label>
    <InputNumber id="bedrooms" class="form-control" @bind-Value="searchCriteria.Bedrooms" />
</div>
<button type="submit" class="btn btn-primary">Search</button>
</EditForm>

@if (isLoading)
{
    <div class="loading-indicator">Loading...</div>
}
else if (errorMessage != null)
{
    <div class="alert alert-danger">@errorMessage</div>
}
else if (properties != null)
{
    <div class="property-grid">
        @foreach (var property in properties)

```

```

    {
        <div class="property-card">
            
            <div class="property-details">
                <h3 class="property-title">@property.Titl
e</h3>
                <p class="property-price">Price: @propert
y.Price.ToString("C")</p>
                <p class="property-location">Location: @p
roperty.Location</p>
                <a href="/property-details/@property.Id"
class="btn btn-primary">View Details</a>
            </div>
        </div>
    }

</div>

<Pagination TotalPages="@totalPages" CurrentPage="@curren
tPage" OnPageChanged="@OnPageChangedAsync" />
}

@code {
    private SearchCriteria searchCriteria = new SearchCriteria();
    private List<Property> properties;
    private bool isLoading;
    private string errorMessage;
    private int currentPage = 1;
    private int totalPages;

    private async Task SearchPropertiesAsync()
    {
        try
        {
            isLoading = true;

```

```

        errorMessage = null;

        var result = await PropertyService.SearchPropertiesAsync(searchCriteria, currentPage);
        properties = result.Properties;
        totalPages = result.TotalPages;
    }
    catch (Exception ex)
    {
        errorMessage = "An error occurred while searching
for properties. Please try again.";
        // Log the exception for further investigation
        Logger.LogError(ex, "Error occurred while searching
for properties.");
    }
    finally
    {
        isLoading = false;
    }
}

private async Task OnPageChangedAsync(int page)
{
    currentPage = page;
    await SearchPropertiesAsync();
}
}

```

## 2. Property Details Component (PropertyDetails.razor)

```

@page "/property-details/{PropertyId:int}"
@inject IPropertyService PropertyService

<h2>Property Details</h2>

```

```

@if (isLoading)
{
    <div class="loading-indicator">Loading...</div>
}
else if (errorMessage != null)
{
    <div class="alert alert-danger">@errorMessage</div>
}
else if (property != null)
{
    <div class="property-details">
        
        <div class="property-info">
            <h3 class="property-title">@property.Title</h3>
            <p class="property-price">Price: @property.Price.ToString("C")</p>
            <p class="property-location">Location: @property.Location</p>
            <p class="property-bedrooms">Bedrooms: @property.Bedrooms</p>
            <p class="property-bathrooms">Bathrooms: @property.Bathrooms</p>
            <p class="property-description">@property.Description</p>
        </div>
    </div>

    <div class="property-actions">
        <button class="btn btn-primary" @onclick="AddToFavoritesAsync">Save to Favorites</button>
        <a href="/property-search" class="btn btn-secondary">Back to Search Results</a>
    </div>

    <div class="property-reviews">

```

```
<!-- Display user reviews and ratings -->
</div>

<div class="related-properties">
    <!-- Display related properties -->
</div>
}

@code {
    [Parameter]
    public int PropertyId { get; set; }

    private Property property;
    private bool isLoading;
    private string errorMessage;

    protected override async Task OnInitializedAsync()
    {
        await LoadPropertyDetailsAsync();
    }

    private async Task LoadPropertyDetailsAsync()
    {
        try
        {
            isLoading = true;
            errorMessage = null;

            property = await PropertyService.GetPropertyByIdA
sync(PropertyId);
        }
        catch (Exception ex)
        {
            errorMessage = "An error occurred while loading p
roperty details. Please try again.";
            // Log the exception for further investigation
        }
    }
}
```

```

        Logger.LogError(ex, "Error occurred while loading
property details.");
    }
    finally
    {
        isLoading = false;
    }
}

private async Task AddToFavoritesAsync()
{
    try
    {
        await PropertyService.AddToFavoritesAsync(property
Id);
        // Show success message or update UI
    }
    catch (Exception ex)
    {
        // Show error message
        // Log the exception for further investigation
        Logger.LogError(ex, "Error occurred while adding
property to favorites.");
    }
}
}

```

### 3. Favorites Component (Favorites.razor)

```

@page "/favorites"
@inject IPropertyService PropertyService

<h2>Favorite Properties</h2>

@if (isLoading)

```

```

{
    <div class="loading-indicator">Loading...</div>
}
else if (errorMessage != null)
{
    <div class="alert alert-danger">@errorMessage</div>
}
else if (favoriteProperties == null || favoriteProperties.Count == 0)
{
    <p>No favorite properties found.</p>
}
else
{
    <div class="favorites-list">
        @foreach (var property in favoriteProperties)
        {
            <div class="favorite-item">
                
                <div class="favorite-details">
                    <h3 class="favorite-title">@property.Title</h3>
                    <p class="favorite-price">Price: @property.Price.ToString("C")</p>
                    <p class="favorite-location">Location: @property.Location</p>
                    <div class="favorite-actions">
                        <button class="btn btn-danger" @onclick="(() => RemoveFromFavoritesAsync(property.Id))">Remove</button>
                        <a href="/property-details/@property.Id" class="btn btn-primary">View Details</a>
                    </div>
                </div>
            </div>
        </div>
    
```

```
        }
    </div>
}

@code {
    private List<Property> favoriteProperties;
    private bool isLoading;
    private string errorMessage;

    protected override async Task OnInitializedAsync()
    {
        await LoadFavoritePropertiesAsync();
    }

    private async Task LoadFavoritePropertiesAsync()
    {
        try
        {
            isLoading = true;
            errorMessage = null;

            favoriteProperties = await PropertyService.GetFavoritePropertiesAsync();
        }
        catch (Exception ex)
        {
            errorMessage = "An error occurred while loading favorite properties. Please try again.";
            // Log the exception for further investigation
            Logger.LogError(ex, "Error occurred while loading favorite properties.");
        }
        finally
        {
            isLoading = false;
        }
    }
}
```

```

    }

    private async Task RemoveFromFavoritesAsync(int propertyId)
    {
        try
        {
            // Show confirmation dialog before removing from favorites
            bool confirmed = await JSRuntime.InvokeAsync<bool>("confirm", "Are you sure you want to remove this property from favorites?");

            if (confirmed)
            {
                await PropertyService.RemoveFromFavoritesAsync(propertyId);
                favoriteProperties = await PropertyService.GetFavoritePropertiesAsync();
            }
        }
        catch (Exception ex)
        {
            // Show error message
            // Log the exception for further investigation
            Logger.LogError(ex, "Error occurred while removing property from favorites.");
        }
    }
}

```

---

These updated components ensure a seamless user experience while integrating error handling and logging for better maintainability and debugging. You can now proceed to implement these updates in your Blazor application. If you have any further questions or need additional features, feel free to ask!

## implementing Backend Services and Connecting Blazor Components

Now that we have updated the Blazor components, the next step is to ensure that the backend services are properly implemented and connected to these components. We will focus on creating the necessary backend services in ASP.NET Core and ensuring they are integrated with the Blazor frontend.

### Tasks:

1. Implement the `IPropertyService` interface and its methods.
2. Implement the `PropertyService` class that interacts with the database and provides data to the Blazor components.
3. Ensure the backend services are properly registered in the `Startup.cs` file for dependency injection.
4. Test the integration between the backend services and the Blazor components.

### 1. Implementing the `IPropertyService` Interface

```
csharpCopy code
// IPropertyService.cs

using System.Collections.Generic;
using System.Threading.Tasks;

public interface IPropertyService
{
    Task<SearchResult> SearchPropertiesAsync(SearchCriteria criteria, int page);
    Task<Property> GetPropertyByIdAsync(int propertyId);
    Task AddToFavoritesAsync(int propertyId);
    Task<List<Property>> GetFavoritePropertiesAsync();
    Task RemoveFromFavoritesAsync(int propertyId);
}
```

## 2. Implementing the **PropertyService** Class

```
csharpCopy code
// PropertyService.cs

using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

public class PropertyService : IPropertyService
{
    private readonly ApplicationDbContext _context;
    private readonly ILogger<PropertyService> _logger;

    public PropertyService(ApplicationDbContext context, ILogger<PropertyService> logger)
    {
        _context = context;
        _logger = logger;
    }

    public async Task<SearchResult> SearchPropertiesAsync(SearchCriteria criteria, int page)
    {
        var query = _context.Properties.AsQueryable();

        if (!string.IsNullOrEmpty(criteria.Location))
        {
            query = query.Where(p => p.Location.Contains(criteria.Location));
        }

        if (criteria.MinPrice.HasValue)
```

```

    {
        query = query.Where(p => p.Price >= criteria.MinPrice.Value);
    }

    if (criteria.MaxPrice.HasValue)
    {
        query = query.Where(p => p.Price <= criteria.MaxPrice.Value);
    }

    if (criteria.Bedrooms.HasValue)
    {
        query = query.Where(p => p.Bedrooms == criteria.Bedrooms.Value);
    }

    var totalItems = await query.CountAsync();
    var properties = await query.Skip((page - 1) * 10).Take(10).ToListAsync();

    return new SearchResult
    {
        Properties = properties,
        TotalPages = (int)Math.Ceiling(totalItems / 10.0)
    };
}

public async Task<Property> GetPropertyByIdAsync(int propertyId)
{
    return await _context.Properties.FindAsync(propertyId);
}

public async Task AddToFavoritesAsync(int propertyId)

```

```

{
    var favorite = new Favorite { PropertyId = propertyId,
        UserId = GetUserId() };
    _context.Favorites.Add(favorite);
    await _context.SaveChangesAsync();
}

public async Task<List<Property>> GetFavoritePropertiesAsync()
{
    var userId = GetUserId();
    return await _context.Favorites
        .Where(f => f.UserId == userId)
        .Include(f => f.Property)
        .Select(f => f.Property)
        .ToListAsync();
}

public async Task RemoveFromFavoritesAsync(int propertyId)
{
    var userId = GetUserId();
    var favorite = await _context.Favorites
        .FirstOrDefaultAsync(f => f.PropertyId == propertyId && f.UserId == userId);

    if (favorite != null)
    {
        _context.Favorites.Remove(favorite);
        await _context.SaveChangesAsync();
    }
}

private string GetUserId()
{
    // Implement logic to retrieve the current user's ID
}

```

```
        return "current-user-id";
    }
}
```

### 3. Registering Services in `Startup.cs`

```
csharpCopy code
// Startup.cs

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

        services.AddScoped<IPROPERTYService, PropertyService>();
        services.AddRazorPages();
        services.AddServerSideBlazor();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
```

```

    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
    });
}

```

## 4. Testing the Integration

Now that the backend services are implemented and registered, we can test the integration with the Blazor components by running the application and performing various actions like searching for properties, viewing property details, and managing favorites.

### Testing Steps:

#### 1. Property Search:

- Navigate to the Property Search page.
- Enter search criteria and submit the form.
- Verify that the search results are displayed correctly.

#### 2. Property Details:

- Click on a property in the search results to view its details.
- Verify that the property details are displayed correctly.

### 3. Favorites Management:

- Add a property to favorites from the property details page.
- Navigate to the Favorites page and verify that the property is listed.
- Remove the property from favorites and verify that it is no longer listed.

## Step-by-Step Implementation

### Step 1: Implement the `IPropertyService` Interface

Create a new file named `IPropertyService.cs`:

```
csharpCopy code
// IPropertyService.cs

using System.Collections.Generic;
using System.Threading.Tasks;

public interface IPropertyService
{
    Task<SearchResult> SearchPropertiesAsync(SearchCriteria criteria, int page);
    Task<Property> GetPropertyByIdAsync(int propertyId);
    Task AddToFavoritesAsync(int propertyId);
    Task<List<Property>> GetFavoritePropertiesAsync();
    Task RemoveFromFavoritesAsync(int propertyId);
}
```

### Step 2: Implement the `PropertyService` Class

Create a new file named `PropertyService.cs`:

```
csharpCopy code
// PropertyService.cs
```

```

using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

public class PropertyService : IPropertyService
{
    private readonly ApplicationDbContext _context;
    private readonly ILogger<PropertyService> _logger;

    public PropertyService(ApplicationDbContext context, ILogger<PropertyService> logger)
    {
        _context = context;
        _logger = logger;
    }

    public async Task<SearchResult> SearchPropertiesAsync
(SearchCriteria criteria, int page)
{
    var query = _context.Properties.AsQueryable();

    if (!string.IsNullOrEmpty(criteria.Location))
    {
        query = query.Where(p => p.Location.Contains(criteria.Location));
    }

    if (criteria.MinPrice.HasValue)
    {
        query = query.Where(p => p.Price >= criteria.MinPrice.Value);
    }
}

```

```

        if (criteria.MaxPrice.HasValue)
        {
            query = query.Where(p => p.Price <= criteria.M
axPrice.Value);
        }

        if (criteria.Bedrooms.HasValue)
        {
            query = query.Where(p => p.Bedrooms == criteri
a.Bedrooms.Value);
        }

        var totalItems = await query.CountAsync();
        var properties = await query.Skip((page - 1) * 1
0).Take(10).ToListAsync();

        return new SearchResult
        {
            Properties = properties,
            TotalPages = (int)Math.Ceiling(totalItems / 1
0.0)
        };
    }

    public async Task<Property> GetPropertyByIdAsync(int p
ropertyId)
    {
        return await _context.Properties.FindAsync(propert
yId);
    }

    public async Task AddToFavoritesAsync(int propertyId)
    {
        var favorite = new Favorite { PropertyId = propert
yId, UserId = GetUserId() };
        _context.Favorites.Add(favorite);
    }
}

```

```

        await _context.SaveChangesAsync();
    }

    public async Task<List<Property>> GetFavoritePropertiesAsync()
    {
        var userId = GetUserId();
        return await _context.Favorites
            .Where(f => f.UserId == userId)
            .Include(f => f.Property)
            .Select(f => f.Property)
            .ToListAsync();
    }

    public async Task RemoveFromFavoritesAsync(int propertyId)
    {
        var userId = GetUserId();
        var favorite = await _context.Favorites
            .FirstOrDefaultAsync(f => f.PropertyId == propertyId && f.UserId == userId);

        if (favorite != null)
        {
            _context.Favorites.Remove(favorite);
            await _context.SaveChangesAsync();
        }
    }

    private string GetUserId()
    {
        // Implement logic to retrieve the current user's ID
        return "current-user-id";
    }

```

```
}
```

### Step 3: Register Services in `Startup.cs`

Modify your `Startup.cs` file to register the `IPropertyService` and `PropertyService` for dependency injection:

```
csharpCopy code
// Startup.cs

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

        services.AddScoped<IPropertyService, PropertyService>();

        services.AddRazorPages();
        services.AddServerSideBlazor();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
```

```

    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
    });
}

```

## Step 4: Testing the Integration

Run your application and test the following functionalities:

### 1. Property Search:

- Navigate to the Property Search page.
- Enter search criteria and submit the form.
- Verify that the search results are displayed correctly.

### 2. Property Details:

- Click on a property in the search results to view its details.
- Verify that the property details are displayed correctly.

### 3. Favorites Management:

- Add a property to favorites from the property details page.
- Navigate to the Favorites page and verify that the property is listed.
- Remove the property from favorites and verify that it is no longer listed.

These steps will ensure that your Blazor frontend components are fully integrated with the ASP.NET Core backend services, providing a seamless user experience.

If you have any specific issues or additional features you'd like to address, please let me know.

## Next Steps: Detailed Implementation of Models, Context, and Pagination

To complete the integration, we need to define the data models, set up the `ApplicationDbContext`, and implement the pagination component.

### Tasks:

1. Define the data models for `Property`, `SearchCriteria`, `SearchResult`, and `Favorite`.
2. Set up the `ApplicationDbContext` for Entity Framework Core.
3. Implement the pagination component.

### 1. Define Data Models

Create a folder named `Models` and add the following model classes:

#### `Property.cs`

```
// Models/Property.cs

public class Property
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Location { get; set; }
    public decimal Price { get; set; }
    public int Bedrooms { get; set; }
    public int Bathrooms { get; set; }
    public string Description { get; set; }
```

```
    public string PhotoUrl { get; set; }  
}
```

### SearchCriteria.cs

```
// Models/SearchCriteria.cs  
  
public class SearchCriteria  
{  
    public string Location { get; set; }  
    public decimal? MinPrice { get; set; }  
    public decimal? MaxPrice { get; set; }  
    public int? Bedrooms { get; set; }  
}
```

### SearchResult.cs

```
// Models/SearchResult.cs  
  
using System.Collections.Generic;  
  
public class SearchResult  
{  
    public List<Property> Properties { get; set; }  
    public int TotalPages { get; set; }  
}
```

### Favorite.cs

```
// Models/Favorite.cs  
  
public class Favorite  
{  
    public int Id { get; set; }  
    public int PropertyId { get; set; }  
    public string UserId { get; set; }
```

```
    public Property Property { get; set; }  
}
```

## 2. Set Up `ApplicationDbContext`

Create a new file named `ApplicationContext.cs`:

```
// ApplicationContext.cs  
  
using Microsoft.EntityFrameworkCore;  
  
public class ApplicationContext : DbContext  
{  
    public ApplicationContext(DbContextOptions<ApplicationDbContext> options)  
        : base(options)  
    {  
    }  
  
    public DbSet<Property> Properties { get; set; }  
    public DbSet<Favorite> Favorites { get; set; }  
}
```

## 3. Implement Pagination Component

Create a new Blazor component named `Pagination.razor`:

### `Pagination.razor`

```
@page "/pagination"  
@using System  
  
@if (TotalPages > 1)  
{  
    <nav aria-label="Page navigation">  
        <ul class="pagination">
```

```

        <li class="page-item @(CurrentPage == 1 ? "disabled" : "")">
            <button class="page-link" @onclick="() =>
OnPageChanged(CurrentPage - 1)">Previous</button>
        </li>

        @for (int i = 1; i <= TotalPages; i++)
        {
            <li class="page-item @(CurrentPage == i ?
"active" : "")">
                <button class="page-link" @onclick="() =>
OnPageChanged(i)">@i</button>
            </li>
        }

        <li class="page-item @(CurrentPage == TotalPages ? "disabled" : "")">
            <button class="page-link" @onclick="() =>
OnPageChanged(CurrentPage + 1)">Next</button>
        </li>
    </ul>
</nav>
}

@code {
    [Parameter]
    public int TotalPages { get; set; }

    [Parameter]
    public int CurrentPage { get; set; }

    [Parameter]
    public EventCallback<int> OnPageChanged { get; set; }
}

```

## Complete Backend Integration

Make sure your `Startup.cs` file is correctly configured as previously mentioned and that your database is set up.

### 1. Migrations and Database Update:

- Use Entity Framework Core to create and update the database.

```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

### 2. Run the Application:

- Start your ASP.NET Core application.
- Test the property search, details, and favorites functionalities to ensure everything is working correctly.

## Summary of Tasks Completed:

1. Implemented `IPropertyService` interface.
2. Created `PropertyService` class.
3. Registered services in `Startup.cs`.
4. Defined data models.
5. Set up `ApplicationDbContext`.
6. Implemented pagination component.
7. Ran migrations and updated the database.

With these steps completed, your Blazor application should be fully integrated with the backend services, providing a comprehensive real estate experience.

If you have further enhancements, features, or issues to address, please let me know!

## Next Steps: Enhancing User Experience and Testing

With the core functionalities implemented, we can now focus on enhancing the user experience and conducting thorough testing. The following tasks will ensure a polished and

user-friendly application:

## Tasks:

1. **Enhance User Interface:** Apply responsive design techniques and improve the overall look and feel of the application.
2. **Implement Error Handling and Notifications:** Ensure the application gracefully handles errors and provides useful feedback to the user.
3. **Testing and Debugging:** Conduct comprehensive testing to identify and fix any issues.
4. **Deploy the Application:** Deploy the application to Azure for real-world testing and user feedback.

### 1. Enhance User Interface

#### Apply Responsive Design

Update your CSS to make the application responsive. You can use Bootstrap classes and custom CSS.

`wwwroot/css/site.css`

```
/* Custom styles for responsive design */
.property-grid {
    display: flex;
    flex-wrap: wrap;
    gap: 20px;
}

.property-card {
    flex: 1 1 calc(33.333% - 20px);
    border: 1px solid #ddd;
    border-radius: 8px;
    overflow: hidden;
    display: flex;
    flex-direction: column;
}
```

```

.property-image {
    width: 100%;
    height: 200px;
    object-fit: cover;
}

.property-details {
    padding: 15px;
    flex-grow: 1;
}

@media (max-width: 768px) {
    .property-card {
        flex: 1 1 calc(50% - 20px);
    }
}

@media (max-width: 576px) {
    .property-card {
        flex: 1 1 100%;
    }
}

```

## 2. Implement Error Handling and Notifications

Use Blazor's built-in features to show notifications and handle errors.

### PropertySearch.razor

```

@page "/property-search"
@inject IPropertyService PropertyService

<h2>Property Search</h2>

<EditForm Model="@searchCriteria" OnValidSubmit="@SearchPropertiesAsync">

```

```

<DataAnnotationsValidator />
<ValidationSummary />

<div class="form-group">
    <label for="location">Location:</label>
    <InputText id="location" class="form-control" @bind-Value="searchCriteria.Location" />
</div>
<div class="form-group">
    <label for="minPrice">Min Price:</label>
    <InputNumber id="minPrice" class="form-control" @bind-Value="searchCriteria.MinPrice" />
</div>
<div class="form-group">
    <label for="maxPrice">Max Price:</label>
    <InputNumber id="maxPrice" class="form-control" @bind-Value="searchCriteria.MaxPrice" />
</div>
<div class="form-group">
    <label for="bedrooms">Bedrooms:</label>
    <InputNumber id="bedrooms" class="form-control" @bind-Value="searchCriteria.Bedrooms" />
</div>
<button type="submit" class="btn btn-primary">Search</button>
</EditForm>

@if (isLoading)
{
    <div class="loading-indicator">Loading...</div>
}
else if (errorMessage != null)
{
    <div class="alert alert-danger">@errorMessage</div>
}
else if (properties != null)

```

```

{
    <div class="property-grid">
        @foreach (var property in properties)
        {
            <div class="property-card">
                
                <div class="property-details">
                    <h3 class="property-title">@property.Title</h3>
                    <p class="property-price">Price: @property.Price.ToString("C")</p>
                    <p class="property-location">Location: @property.Location</p>
                    <a href="/property-details/@property.Id" class="btn btn-primary">View Details</a>
                </div>
            </div>
        }
    </div>

    <Pagination TotalPages="@totalPages" CurrentPage="@currentPage" OnPageChanged="@OnPageChangedAsync" />
}

@code {
    private SearchCriteria searchCriteria = new SearchCriteria();
    private List<Property> properties;
    private bool isLoading;
    private string errorMessage;
    private int currentPage = 1;
    private int totalPages;

    private async Task SearchPropertiesAsync()
    {
}

```

```

        try
        {
            isLoading = true;
            errorMessage = null;

            var result = await PropertyService.SearchPropertiesAsync(searchCriteria, currentPage);
            properties = result.Properties;
            totalPages = result.TotalPages;
        }
        catch (Exception ex)
        {
            errorMessage = "An error occurred while searching for properties. Please try again.";
            Logger.LogError(ex, "Error occurred while searching for properties.");
        }
        finally
        {
            isLoading = false;
        }
    }

    private async Task OnPageChangedAsync(int page)
    {
        currentPage = page;
        await SearchPropertiesAsync();
    }
}

```

### 3. Testing and Debugging

Perform comprehensive testing including unit tests, integration tests, and user acceptance testing (UAT).

#### Unit Tests

Create unit tests for your service methods using xUnit or NUnit.

### PropertyServiceTests.cs

```
// PropertyServiceTests.cs

using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using Moq;
using Xunit;

public class PropertyServiceTests
{
    private readonly Mock<ILogger<PropertyService>> _loggerMock;
    private readonly ApplicationDbContext _context;
    private readonly PropertyService _propertyService;

    public PropertyServiceTests()
    {
        var options = new DbContextOptionsBuilder<ApplicationDbContext>()
            .UseInMemoryDatabase(databaseName: "TestDatabase")
            .Options;

        _context = new ApplicationDbContext(options);
        _loggerMock = new Mock<ILogger<PropertyService>>();
        _propertyService = new PropertyService(_context, _loggerMock.Object);
    }

    [Fact]
```

```

public async Task SearchPropertiesAsync_ReturnsCorrect
Results()
{
    // Arrange
    var properties = new List<Property>
    {
        new Property { Id = 1, Title = "Test Property
1", Location = "Location1", Price = 100000 },
        new Property { Id = 2, Title = "Test Property
2", Location = "Location2", Price = 200000 }
    };

    await _context.Properties.AddRangeAsync(properties);
    await _context.SaveChangesAsync();

    var criteria = new SearchCriteria { Location = "Lo
cation1" };

    // Act
    var result = await _propertyService.SearchProperti
esAsync(criteria, 1);

    // Assert
    Assert.Single(result.Properties);
    Assert.Equal("Test Property 1", result.Properties
[0].Title);
}
}

```

## Integration Tests

Test the integration between Blazor components and backend services.

### **PropertySearchIntegrationTests.cs**

```

// PropertySearchIntegrationTests.cs

using Microsoft.AspNetCore.Mvc.Testing;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

public class PropertySearchIntegrationTests : IClassFixture<WebApplicationFactory<Startup>>
{
    private readonly HttpClient _client;

    public PropertySearchIntegrationTests(WebApplicationFactory<Startup> factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task PropertySearch>ReturnsResults()
    {
        // Arrange
        var response = await _client.GetAsync("/property-search");

        // Act
        var responseString = await response.Content.ReadAsStringAsync();

        // Assert
        Assert.Contains("Property Search", responseString);
    }
}

```

## **4. Deploy the Application**

Deploy your application to Azure for real-world testing and user feedback.

### **Steps for Deployment:**

#### **1. Create an Azure App Service:**

- Use the Azure Portal to create a new App Service instance.

#### **2. Configure the Database:**

- Set up an Azure SQL Database and configure connection strings.

#### **3. Deploy the Application:**

- Use Visual Studio or Azure DevOps to deploy your application to the App Service.

#### **4. Monitor and Optimize:**

- Use Azure Monitor and Application Insights to monitor performance and optimize the application.
- 

### **Summary of Tasks Completed:**

1. Enhanced user interface with responsive design.
2. Implemented error handling and notifications.
3. Conducted unit and integration testing.
4. Prepared the application for deployment to Azure.

With these steps completed, your NexHomeAgent application should provide a polished and user-friendly experience, ready for real-world use and feedback.

If you have any further enhancements, features, or issues to address, please let me know!

To set up the front end in Visual Studio, particularly for a Blazor WebAssembly project, you can follow these steps:

### **Step 1: Create a New Blazor WebAssembly Project**

1. Open Visual Studio.
2. Create a new project.

- Select **Blazor WebAssembly App** from the project templates.
- Click **Next**.

### 3. Configure your new project.

- Name your project, choose a location, and solution name.
- Click **Next**.

### 4. Additional Information.

- Select **.NET 6.0 (Long-term support)** as the target framework.
- Ensure **ASP.NET Core hosted** is checked if you want a full-stack application with a backend.
- Click **Create**.

## Step 2: Add npm Support for Frontend Dependencies

### 1. Add npm Configuration File (`package.json`).

- Right-click on the project in Solution Explorer.
- Select **Add > New Item**.
- Choose **npm Configuration File** and name it `package.json`.

### 2. Configure `package.json` with Frontend Dependencies.

- Open `package.json` and add the following configuration:

```
{
  "name": "BlazorApp",
  "version": "1.0.0",
  "description": "A Blazor WebAssembly application",
  "scripts": {
    "build:css": "npx tailwindcss build src/tailwind.css -o wwwroot/css/tailwind.css",
    "build": "npm run build:css"
  },
  "dependencies": {
    "bootstrap": "^5.1.0",
  }
}
```

```
    "tailwindcss": "^2.2.19"
},
"devDependencies": {
    "terser-webpack-plugin": "^5.1.4",
    "css-minimizer-webpack-plugin": "^3.0.2"
}
}
```

## Step 3: Add Tailwind CSS Configuration

### 1. Initialize Tailwind CSS Configuration.

- Open a terminal in the project directory.
- Run the following command to initialize the Tailwind CSS configuration file:

```
npx tailwindcss init
```

### 1. Configure `tailwind.config.js`.

- Open the newly created `tailwind.config.js` and configure it:

```
module.exports = {
  purge: ['./**/*.{razor,html}'],
  darkMode: false,
  theme: {
    extend: {},
  },
  variants: {
    extend: {},
  },
  plugins: [],
};
```

### 1. Create Tailwind CSS Entry Point.

- Create a file named `src/tailwind.css` and add the Tailwind directives:

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;
```

## Step 4: Automate CSS Build Process

### 1. Modify `.csproj` to Include npm Commands.

- Open your `.csproj` file and add the following `Target` elements to automate npm installation and CSS build:

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">  
  
<PropertyGroup>  
    <TargetFramework>net6.0</TargetFramework>  
    <OutputType>Exe</OutputType>  
    <Nullable>enable</Nullable>  
    <ImplicitUsings>enable</ImplicitUsings>  
</PropertyGroup>  
  
<ItemGroup>  
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly" Version="6.0.0" />  
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer" Version="6.0.0" PrivateAssets="all" />  
</ItemGroup>  
  
<Target Name="NpmInstall" BeforeTargets="Build">  
    <Exec Command="npm install" />  
</Target>  
  
<Target Name="TailwindBuild" AfterTargets="Build">  
    <Exec Command="npm run build:css" />  
</Target>
```

```
</Project>
```

## Step 5: Include CSS in the Project

### 1. Reference CSS Files in `index.html` or `_Host.cshtml`.

- Open `wwwroot/index.html` (or `_Host.cshtml` if server-side Blazor) and include the Bootstrap and Tailwind CSS files:

```
<!DOCTYPE html>
<html>
<head>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
    <link href="css/tailwind.css" rel="stylesheet" />
    <title>BlazorApp</title>
</head>
<body>
    <app>Loading...</app>
    <script src="_framework/blazor.webassembly.js"></script>
</body>
</html>
```

## Step 6: Create a Script to Install Dependencies

### 1. Create a PowerShell Script to Install npm Dependencies.

- Create a file named `install-dependencies.ps1` with the following content:

```
# PowerShell script to install npm dependencies
Write-Output "Installing npm dependencies..."

# Navigate to the project directory
Set-Location -Path $PSScriptRoot

# Run npm install
```

```
npm install  
  
Write-Output "Dependencies installed successfully."
```

## Step 7: Run the Script and Build the Project

### 1. Run the PowerShell Script to Install npm Dependencies:

```
.\\install-dependencies.ps1
```

### 1. Build Your Project in Visual Studio.

- Press **F5** to build and run your project.
- The `NpmInstall` and `TailwindBuild` targets will ensure that npm dependencies are installed and Tailwind CSS is built before the Blazor app is built.

By following these steps, you will have a Blazor WebAssembly project set up in Visual Studio with npm support, Bootstrap, and Tailwind CSS integrated, and a script to install all front-end dependencies.



# Setting Up the Development Environment in Visual Studio

## Development Environment in Visual Studio

### Setting Up the Development Environment in Visual Studio

This guide will walk you through setting up the Dwellingly | AI application development environment in Visual Studio, configuring Azure resources, and preparing for local development and testing.

### Prerequisites

- Visual Studio 2022
- Azure CLI
- .NET 8 SDK
- Azure Subscription

### Step-by-Step Guide

## Step 1: Install Prerequisites

### 1. Install Visual Studio 2022:

- Download and install Visual Studio 2022 from the [Visual Studio website](#).
- Select the following workloads during installation:
  - [ASP.NET](#) and web development
  - Azure development

### 2. Install Azure CLI:

- Follow the instructions from the [official Azure documentation](#).
- Sign in to your Azure account:

```
az login
```

### 3. Install .NET SDK:

- Download and install the .NET 8 SDK from the [.NET website](#).

## Step 2: Create the Solution and Projects in Visual Studio

### 1. Create a New Solution:

- Open Visual Studio 2022.
- Go to `File -> New -> Project`.
- Select `Blank Solution`.
- Name it `Dwellingly`.
- Choose a location for the solution and click `Create`.

### 2. Add Projects to the Solution:

- Right-click on the `Dwellingly` solution in the Solution Explorer -> `Add -> New Project`.

#### Backend Project:

- Select `ASP.NET Core Web API`.
- Name it `Dwellingly.API`.

### **Frontend Project:**

- Select `Blazor WebAssembly App`.
- Name it `Dwellingly.Client`.

### **Middleware/Business Logic Project:**

- Select `Class Library (.NET Standard)`.
- Name it `Dwellingly.Business`.

### **Data Access Project:**

- Select `Class Library (.NET Standard)`.
- Name it `Dwellingly.Data`.

## **Step 3: Configure Each Project**

### **3.1 Backend Project (`Dwellingly.API`)**

#### **1. Install Required NuGet Packages:**

- Open the `NuGet Package Manager Console`.
- Select `Dwellingly.API` as the Default Project.
- Install the following packages:

```
Install-Package Microsoft.EntityFrameworkCore  
Install-Package Microsoft.EntityFrameworkCore.SqlServer  
Install-Package Dapper  
Install-Package AutoMapper  
Install-Package Microsoft.Data.SqlClient
```

#### **2. Set Up Entity Framework Core:**

- Add a `DbContext` class in the `Dwellingly.API` project:

```
using Microsoft.EntityFrameworkCore;  
  
namespace Dwellingly.Data  
{
```

```

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {

    }

    public DbSet<Property> Properties { get; set; }
}

public class Property
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public decimal Price { get; set; }
}

```

- Configure the connection string in `appsettings.json`:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\\\\\mssqllocaldb;Database=DwellinglyDB;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
}
```

```
        "AllowedHosts": "*"
    }
```

- Update `Program.cs` to register the `DbContext`:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[]
        args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}

public class Startup
{
    public IConfiguration Configuration { get; }

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
```

```

        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}

```

## 3.2 Frontend Project ( Dwellingly.Client )

### 1. Install Required NuGet Packages:

- Open the `NuGet Package Manager Console`.
- Select `Dwellingly.Client` as the Default Project.
- Install the following packages:

```
Install-Package Microsoft.AspNetCore.Components.WebAssembly
Install-Package Blazored.LocalStorage
```

## 2. Set Up HttpClient:

- Update `Program.cs` to configure `HttpClient`:

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.RootComponents.Add<App>("#app");

        builder.Services.AddScoped(sp => new HttpClient
        { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
        builder.Services.AddBlazoredLocalStorage();

        await builder.Build().RunAsync();
    }
}
```

## 3.3 Middleware/Business Logic Project (`Dwellingly.Business`)

### 1. Create Business Logic Classes:

- Create necessary services and business logic in this project.
- For example, create a service to manage properties:

```

using Dwellingly.Data;

namespace Dwellingly.Business
{
    public class PropertyService
    {
        private readonly ApplicationContext _context;

        public PropertyService(ApplicationContext context)
        {
            _context = context;
        }

        public async Task<List<Property>> GetProperties
Async()
        {
            return await _context.Properties.ToListAsyn
c();
        }
    }
}

```

### 3.4 Data Access Project ( [Dwellingly.Data](#) )

#### 1. Add Data Entities:

- Create entities and repositories in this project.
- For example, create a [Property](#) entity:

```

namespace Dwellingly.Data
{
    public class Property
    {
        public int Id { get; set; }

```

```
        public string Name { get; set; }
        public string Address { get; set; }
        public decimal Price { get; set; }
    }
}
```

## Step 4: Configure Azure Resources for Development

### 1. Log in to Azure using the Azure CLI:

```
az login
```

### 2. Create a Resource Group:

```
az group create --name DwellinglyDevResourceGroup --location eastus
```

### 3. Create an Azure SQL Database and Server:

```
az sql server create --name dwellingly-dev-sql-server --resource-group DwellinglyDevResourceGroup --location eastus --admin-user sqladmin --admin-password YourStrong(!)Password
```

```
az sql db create --resource-group DwellinglyDevResourceGroup --server dwellingly-dev-sql-server --name DwellinglyDevDB --service-objective S0
```

### 4. Create Azure Container Registry (ACR):

```
az acr create --resource-group DwellinglyDevResourceGroup --name DwellinglyDevACR --sku Basic
```

### 5. Create an Azure Kubernetes Service (AKS) Cluster with ACR Integration:

```
az aks create --resource-group DwellinglyDevResourceGroup  
--name DwellinglyDevAKSCluster --node-count 1 --enable-addons monitoring --generate-ssh-keys --attach-acr DwellinglyDevACR
```

#### 6. Get AKS Credentials:

```
az aks get-credentials --resource-group DwellinglyDevResourceGroup --name DwellinglyDevAKSCluster
```

#### 7. Log in to ACR:

```
az acr login --name DwellinglyDevACR
```

## Step 5: Build and Deploy the Application for Development

#### 1. Build the Application:

- Open the `NuGet Package Manager Console` and build the solution.
- Verify that the projects build successfully.

#### 2. Create a Dockerfile in the `Dwellingly.API` Project:

- Add a `Dockerfile` to the root of the `Dwellingly.API` project:

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base  
WORKDIR /app  
EXPOSE 80  
  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build  
WORKDIR /src
```

## Step 5: Build and Deploy the Application for Development (continued)

#### 1. Create a Dockerfile in the `Dwellingly.API` Project (continued):

- Add a `Dockerfile` to the root of the `Dwellingly.API` project:

```

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["Dwellingly.API/Dwellingly.API.csproj", "Dwellingly.API/"]
RUN dotnet restore "Dwellingly.API/Dwellingly.API.csproj"
COPY . .
WORKDIR "/src/Dwellingly.API"
RUN dotnet build "Dwellingly.API.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "Dwellingly.API.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Dwellingly.API.dll"]

```

## 2. Build and Tag the Docker Image:

- Open a terminal and navigate to the `Dwellingly.API` project directory.
- Build the Docker image:

```
docker build -t dwellingly-dev-api .
```

- Tag the Docker image for ACR:

```
docker tag dwellingly-dev-api dwellinglydevacr.azurecr.
```

```
io/dwellingly-dev-api:latest
```

### 3. Push the Docker Image to ACR:

- Push the image to ACR:

```
docker push dwellinglydevacr.azurecr.io/dwellingly-dev-
api:latest
```

### 4. Create Kubernetes Deployment and Service YAML Files:

- Create `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dwellingly-dev-api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dwellingly-dev-api
  template:
    metadata:
      labels:
        app: dwellingly-dev-api
    spec:
      containers:
        - name: dwellingly-dev-api
          image: dwellinglydevacr.azurecr.io/dwellingly-d
ev-api:latest
      ports:
        - containerPort: 80
```

- Create `service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: dwellingly-dev-api
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: dwellingly-dev-api
```

## 5. Deploy to AKS:

- Apply the deployment and service configurations:

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

## 6. Verify the Deployment:

- Check the status of the deployment:

```
kubectl get deployments
```

- Check the status of the service:

```
kubectl get services
```

- Get the external IP of the service:

```
kubectl get service dwellingly-dev-api
```

# Step 6: Setting Up DevOps CI/CD in Azure DevOps

## 1. Set Up Azure DevOps Project:

- Create a new project in Azure DevOps.

## 2. Create CI Pipeline:

- Go to Pipelines -> Create Pipeline.
- Select the repository where your code is hosted.
- Configure the pipeline with the following YAML:

```

trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '8.x'
    installationPath: $(Agent.ToolsDirectory)/dotnet

- script: dotnet build --configuration Release
  displayName: 'Build project'

- task: PublishBuildArtifacts@1
  inputs:
    PathToPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'drop'
    publishLocation: 'Container'

```

## 3. Create CD Pipeline:

- Go to Releases -> New pipeline.
- Configure the stages for deployment (e.g., Dev, Staging, Prod).
- Set up deployment tasks to deploy the Docker image to AKS.

## Step 7: Integrate User Data and Azure OpenAI for the Chatbot

### 1. Configure Azure SQL Database for User Data:

- Use Entity Framework Core to interact with Azure SQL Database as shown in the backend configuration.

### 2. Integrate Azure OpenAI Service:

- Sign up for Azure OpenAI Service and get your API key.
- Add a service to call Azure OpenAI in your backend project:

```
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json;

namespace Dwellingly.API.Services
{
    public class OpenAIService
    {
        private readonly HttpClient _httpClient;
        private readonly string _apiKey = "YOUR_OPENAI_
API_KEY";

        public OpenAIService(HttpClient httpClient)
        {
            _httpClient = httpClient;
        }

        public async Task<string> GetChatbotResponse(st
ring prompt)
        {
            var requestContent = new StringContent(Json
Convert.SerializeObject(new
            {
                prompt = prompt,
            })
        }
    }
}
```

```

        max_tokens = 150
    }, Encoding.UTF8, "application/json");

        _httpClient.DefaultRequestHeaders.Authorization =
            new AuthenticationHeaderValue("Bearer", _apiKey);
    }

        var response = await _httpClient.PostAsync(
            "<https://api.openai.com/v1/engines/davinci-codex/completions>",
            requestContent);
        response.EnsureSuccessStatusCode();

        var responseContent = await response.Content.ReadAsStringAsync();
        var result = JsonConvert.DeserializeObject<
            OpenAIResponse>(responseContent);

        return result.Choices.FirstOrDefault()?.Text.Trim();
    }
}

public class OpenAIResponse
{
    public List<Choice> Choices { get; set; }
}

public class Choice
{
    public string Text { get; set; }
}

```

### 3. Integrate Chatbot in Frontend:

- Add a chat component to your Blazor app that interacts with the OpenAI service.

```

@page "/chat"

<h3>Chat with our AI</h3>

<div>
    <input @bind="userInput" placeholder="Type your message" />
    <button @onclick="SendMessage">Send</button>
</div>

<div>
    <p>@response</p>
</div>

@code {
    private string userInput = string.Empty;
    private string response = string.Empty;

    private async Task SendMessage()
    {
        var openAIService = new OpenAIService(new HttpClient());
        response = await openAIService.GetChatbotResponse(userInput);
    }
}

```

## Conclusion

Following these steps will set up the development environment, create and run the Dwellingly | AI application locally, configure Azure resources for development, and integrate Azure OpenAI for the chatbot functionality. This comprehensive guide ensures that the Dwellingly | AI application is well-structured, scalable, and integrated with modern development practices.

## ReadMe Instructions

# **ReadMe for Development Environment Setup and Container Deployment for Dwellingly | AI**

## **Overview**

This guide provides detailed instructions to set up the development environment and deploy the Dwellingly | AI application using Azure Kubernetes Service (AKS) and Azure Container Registry (ACR) without using Docker for deployment. Instead, Azure-native services are used to build and deploy the application.

## **Prerequisites**

- Azure CLI installed
- kubectl CLI installed
- .NET 8 SDK
- Azure Subscription

## **Steps**

### **1. Log in to Azure:**

```
shCopy code  
az login
```

### **2. Create a resource group:**

```
shCopy code  
az group create --name DwellinglyResourceGroup --location  
eastus
```

### **3. Create an Azure Container Registry (ACR):**

```
shCopy code  
az acr create --resource-group DwellinglyResourceGroup --n
```

```
ame DwellinglyACR --sku Basic
```

#### 4. Create an AKS Cluster with ACR integration:

```
shCopy code  
az aks create --resource-group DwellinglyResourceGroup --n  
ame DwellinglyAKSCluster --node-count 1 --enable-addons mo  
nitoring --generate-ssh-keys --attach-acr DwellinglyACR
```

#### 5. Get AKS credentials:

```
shCopy code  
az aks get-credentials --resource-group DwellinglyResource  
Group --name DwellinglyAKSCluster
```

#### 6. Log in to ACR:

```
shCopy code  
az acr login --name DwellinglyACR
```

#### 7. Create a Dockerfile in your project root (this step uses Docker only for building the image):

```
dockerfileCopy code  
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base  
WORKDIR /app  
EXPOSE 80  
  
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build  
WORKDIR /src  
COPY ["Dwellingly.API/Dwellingly.API.csproj", "Dwellingly.
```

```
API/"]  
RUN dotnet restore "Dwellingly.API/Dwellingly.API.csproj"  
COPY . .  
WORKDIR "/src/Dwellingly.API"  
RUN dotnet build "Dwellingly.API.csproj" -c Release -o /app/build  
  
FROM build AS publish  
RUN dotnet publish "Dwellingly.API.csproj" -c Release -o /app/publish  
  
FROM base AS final  
WORKDIR /app  
COPY --from=publish /app/publish .  
ENTRYPOINT ["dotnet", "Dwellingly.API.dll"]
```

#### 8. Build the application image:

```
shCopy code  
docker build -t dwellingly-api .
```

#### 9. Tag the application image for ACR:

```
shCopy code  
docker tag dwellingly-api dwellinglyacr.azurecr.io/dwellingly-api:latest
```

#### 10. Push the application image to ACR:

```
shCopy code  
docker push dwellinglyacr.azurecr.io/dwellingly-api:latest
```

**11. Create a Kubernetes deployment YAML file (`deployment.yaml`):**

```
yamlCopy code
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dwellingly-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: dwellingly-api
  template:
    metadata:
      labels:
        app: dwellingly-api
    spec:
      containers:
        - name: dwellingly-api
          image: dwellinglyacr.azurecr.io/dwellingly-api:latest
      ports:
        - containerPort: 80
```

**12. Create a Kubernetes service YAML file (`service.yaml`):**

```
yamlCopy code
apiVersion: v1
kind: Service
metadata:
  name: dwellingly-api
spec:
  type: LoadBalancer
  ports:
```

```
- port: 80
  targetPort: 80
  selector:
    app: dwellingly-api
```

### 13. Deploy to AKS:

```
shCopy code
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
kubectl get deployments
kubectl get services
kubectl get service dwellingly-api
```

## Conclusion

This guide provides all necessary steps to set up the development environment, create and run a .NET application image for your ASP.NET Core API, and deploy the containerized application to Azure Kubernetes Service, ensuring a robust and scalable infrastructure for the Dwellingly | AI application using Azure-native services.

create a script to install the pre requirements

Here is a script to install the prerequisites for setting up the Dwellingly | AI application on Azure Kubernetes Service (AKS) and Azure Container Registry (ACR). This script will:

1. Install Azure CLI
2. Install kubectl
3. Install .NET SDK
4. Log in to Azure

## Prerequisites Installation Script

```
bashCopy code
#!/bin/bash

# Exit script on any error
set -e

# Update package list and install dependencies
sudo apt update
sudo apt install -y apt-transport-https ca-certificates curl
software-properties-common

# Install Azure CLI
echo "Installing Azure CLI..."
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

# Verify Azure CLI installation
az --version

# Install kubectl
echo "Installing kubectl..."
sudo az aks install-cli

# Verify kubectl installation
kubectl version --client

# Install .NET SDK
echo "Installing .NET SDK..."
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb

sudo apt update
sudo apt install -y dotnet-sdk-8.0
```

```
# Verify .NET SDK installation
dotnet --version

# Log in to Azure
echo "Logging in to Azure..."
az login

echo "All prerequisites installed successfully."
```

## Usage Instructions

### 1. Save the Script:

- Save the script as `install_prerequisites.sh`.

### 2. Make the Script Executable:

- Run the following command to make the script executable:

```
shCopy code
chmod +x install_prerequisites.sh
```

### 3. Run the Script:

- Execute the script to install all prerequisites:

```
shCopy code
./install_prerequisites.sh
```

This script will install the necessary tools and log you into Azure. Once the script completes, you will have Azure CLI, kubectl, and .NET SDK installed and ready to use.

## Improved Setup Instructions for Dwellingly | AI

## Overview

To set up the development environment and deploy the Dwellingly | AI application using Docker and Azure, follow these steps. This guide includes installing necessary tools, setting up the development environment, containerizing the application, and deploying it to Azure.

## Prerequisites

- Visual Studio 2022
- Docker
- Azure CLI
- .NET 8 SDK
- Azure Subscription

## Detailed Instructions

### Step 1: Install Visual Studio 2022

#### 1. Download and Install Visual Studio 2022:

- Visit the [Visual Studio website](#).
- Download and install the Community, Professional, or Enterprise edition.
- During installation, select the following workloads:
  - **ASP.NET and web development**
  - **Azure development**
  - **Data storage and processing**

### Step 2: Configure Azure CLI

#### 1. Download and Install Azure CLI:

- Follow the instructions from the [official Azure documentation](#).

#### 2. Sign in to your Azure account:

```
shCopy code  
az login
```

## Step 3: Set Up Git

### 1. Download and Install Git:

- Visit the [official Git website](#).

### 2. Configure Git with your user information:

```
shCopy code  
git config --global user.name "Your Name"  
git config --global user.email "your.email@example.com"
```

## Step 4: Set Up Docker

### 1. Download and Install Docker Desktop:

- Visit the Docker website and follow the installation instructions for your operating system.

### 2. Verify Docker Installation:

```
shCopy code  
docker --version
```

## Step 5: Create ASP.NET Core Project

### 1. Open Visual Studio 2022.

### 2. Create a New Project:

- Select **ASP.NET Core Web API**.
- Choose **.NET 8.0** as the target framework.
- Name the project **Dwellingly.API**.

## Step 6: Install Required NuGet Packages

**1. Open the NuGet Package Manager Console.**

**2. Install the following packages:**

```
shCopy code
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Dapper
Install-Package AutoMapper
Install-Package Microsoft.Data.SqlClient
```

## Step 7: Configure Entity Framework Core

**1. Add a DbContext class:**

```
csharpCopy code
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {

    }

    public DbSet<Property> Properties { get; set; }
}
```

**2. Configure the connection string in `appsettings.json`:**

```
jsonCopy code
"ConnectionStrings": {
    "DefaultConnection": "Server=tcp:<your_server>.database.windows.net,1433;Initial Catalog=<your_db>;Persist Security Info=False;User ID=<your_user>;Password=<your_password>;"
```

```
MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"  
}
```

### 3. Register the DbContext in `Program.cs` :

```
csharpCopy code  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));  
    services.AddControllers();  
}
```

## Step 8: Create API Endpoints

### 1. Create a controller for property listings:

```
csharpCopy code  
[ApiController]  
[Route("api/[controller]")]  
public class PropertiesController : ControllerBase  
{  
    private readonly ApplicationContext _context;  
  
    public PropertiesController(ApplicationContext context)  
    {  
        _context = context;  
    }  
  
    [HttpGet]  
    public async Task<ActionResult<IEnumerable<Property>>>
```

```
    GetProperties()
    {
        return await _context.Properties.ToListAsync();
    }
}
```

## Step 9: Create Dockerfile for the API

1. Add a **Dockerfile** to the root of the **Dwellingly.API** project:

```
dockerfileCopy code
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["Dwellingly.API/Dwellingly.API.csproj", "Dwellingly.API/"]
RUN dotnet restore "Dwellingly.API/Dwellingly.API.csproj"
COPY . .
WORKDIR "/src/Dwellingly.API"
RUN dotnet build "Dwellingly.API.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "Dwellingly.API.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Dwellingly.API.dll"]
```

## Step 10: Build and Run Docker Image Locally

1. Open a terminal and navigate to the project directory.
2. Build the Docker image:

```
shCopy code
docker build -t dwellingly-api .
```

3. Run the Docker container:

```
shCopy code
docker run -d -p 8080:80 --name dwellingly-api-container dwellingly-api
```

4. Verify the application is running by navigating to <http://localhost:8080/api/properties> in your browser.

## Step 11: Push Docker Image to Azure Container Registry (ACR)

1. Login to ACR:

```
shCopy code
az acr login --name <your_acr_name>
```

2. Tag the Docker image:

```
shCopy code
docker tag dwellingly-api <your_acr_name>.azurecr.io/dwellingly-api:latest
```

3. Push the image to ACR:

```
shCopy code  
docker push <your_acr_name>.azurecr.io/dwellingly-api:latest
```

## Step 12: Create Azure App Service and Deploy Container

### 1. Create a resource group:

```
shCopy code  
az group create --name DwellinglyResourceGroup --location eastus
```

### 2. Create an Azure App Service plan:

```
shCopy code  
az appservice plan create --name DwellinglyAppServicePlan  
--resource-group DwellinglyResourceGroup --sku B1 --is-lin  
ux
```

### 3. Create a Web App for Containers:

```
shCopy code  
az webapp create --resource-group DwellinglyResourceGroup  
--plan DwellinglyAppServicePlan --name DwellinglyWebApp --  
deployment-container-image-name <your_acr_name>.azurecr.i  
o/dwellingly-api:latest
```

### 4. Configure the Web App to pull from ACR:

```
shCopy code  
az webapp config container set --name DwellinglyWebApp --r
```

```
resource-group DwellinglyResourceGroup --docker-custom-image-name <your_acr_name>.azurecr.io/dwellingly-api:latest --docker-registry-server-url https://<your_acr_name>.azurecr.io
```

## Conclusion

Following these steps will set up your development environment, create and run a Docker container for your ASP.NET Core API, and deploy the containerized application to Azure App Service. This setup ensures a robust and scalable infrastructure for the Dwellingly | AI application.

## ReadMe Instructions

### **ReadMe for Development Environment Setup and Container Deployment for Dwellingly | AI**

## Overview

This guide provides detailed instructions to set up the development environment and deploy the Dwellingly | AI application using Docker and Azure. It covers installing necessary tools, setting up the development environment, containerizing the application, and deploying it to Azure.

## Prerequisites

- Visual Studio 2022
- Docker
- Azure CLI
- .NET 8 SDK
- Azure Subscription

## Steps

### 1. Install Visual Studio 2022

- Download and install Visual Studio 2022 from the [Visual Studio website](#).
- Select the workloads: ASP.NET and web development, Azure development, Data storage and processing.

## 2. Configure Azure CLI

- Download and install the Azure CLI from the [official Azure documentation](#).
- Sign in to your Azure account: `az login`.

## 3. Set Up Git

- Download and install Git from the [official Git website](#).
- Configure Git:

```
shCopy code
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

## 4. Set Up Docker

- Download and install Docker Desktop from the Docker website.
- Verify Docker installation: `docker --version`.

## 5. Create ASP.NET Core Project

- Open Visual Studio 2022.
- Create a new project: ASP.NET Core Web API, .NET 8.0, named `Dwellingly.API`.

## 6. Install Required NuGet Packages

- Open the NuGet Package Manager Console.
- Install packages:

```
shCopy code
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Dapper
```

```
Install-Package AutoMapper  
Install-Package Microsoft.Data.SqlClient
```

## 7. Configure Entity Framework Core

- Add a `DbContext` class.
- Configure the connection string in `appsettings.json`.
- Register the `DbContext` in `Program.cs`.

## 8. Create API Endpoints

- Create a controller for property listings.

## 9. Create Dockerfile for the API

- Add a `Dockerfile` to the root of the `Dwellingly.API` project.

## 10. Build and Run Docker Image Locally

- Build the Docker image: `docker build -t dwellingly-api .`
- Run the Docker container: `docker run -d -p 8080:80 --name dwellingly-api-container dwellingly-api`
- Verify the application at `http://localhost:8080/api/properties`.

## 11. Push Docker Image to Azure Container Registry (ACR)

- Login to ACR: `az acr login --name <your_acr_name>`
- Tag the Docker image: `docker tag dwellingly-api <your_acr_name>.azurecr.io/dwellingly-api:latest`
- Push the image to ACR: `docker push <your_acr_name>.azurecr.io/dwellingly-api:latest`

## 12. Create Azure App Service and Deploy Container

- Create a resource group: `az group create --name DwellinglyResourceGroup --location eastus`
- Create an Azure App Service plan: `az appservice plan create --name DwellinglyAppServicePlan --resource-group DwellinglyResourceGroup --sku B1 --is-linux`
- Create a Web App for Containers: `az webapp create --resource-group DwellinglyResourceGroup --plan DwellinglyAppServicePlan --name DwellinglyWebApp --`

```
deployment-container-image-name <your_acr_name>.azurecr.io/dwellingly-api:latest
```

- Configure the Web App to pull from ACR:

```
shCopy code  
az webapp config container set --name DwellinglyWebApp  
--resource-group DwellinglyResourceGroup --docker-custo  
m-image-name <your_acr_name>.azurecr.io/dwellingly-api:  
latest --docker-registry-server-url https://<your_acr_n  
ame>.azurecr.io
```

## Conclusion

This guide provides all necessary steps to set up the development environment, create and run a Docker container for your ASP.NET Core API, and deploy the containerized application to Azure App Service, ensuring a robust and scalable infrastructure for the Dwellingly | AI application.

## Save Files

Let's save these instructions into separate files for download.

1. Backend Development Instructions
2. Frontend Development Instructions
3. AI/ML Integration Instructions
4. DevOps Instructions
5. ReadMe Instructions

Feel free to download each of these files to access the detailed instructions.

Here is a comprehensive script to set up the environment, including creating the necessary Azure resources, configuring them, and deploying the Dwellingly | AI application.

## Azure Resource Setup Script

```

#!/bin/bash

# Exit script on any error
set -e

# Variables
RESOURCE_GROUP="DwellinglyResourceGroup"
LOCATION="eastus"
ACR_NAME="DwellinglyACR"
AKS_NAME="DwellinglyAKSCluster"
ACR_URL="$ACR_NAME.azurecr.io"
DOCKER_IMAGE_NAME="dwellingly-api"
DOCKER_IMAGE_TAG="latest"
APP_NAME="Dwellingly.API"
SQL_SERVER_NAME="dwellingly-sql-server"
SQL_DATABASE_NAME="DwellinglyDB"
SQL_ADMIN_USER="sqladmin"
SQL_ADMIN_PASSWORD="YourStrong(!)Password"
KEY_VAULT_NAME="DwellinglyKeyVault"

# Log in to Azure
echo "Logging in to Azure..."
az login

# Create resource group
echo "Creating resource group..."
az group create --name $RESOURCE_GROUP --location $LOCATION

# Create Azure Container Registry (ACR)
echo "Creating Azure Container Registry..."
az acr create --resource-group $RESOURCE_GROUP --name $ACR_NAME --sku Basic

# Create AKS Cluster with ACR integration
echo "Creating Azure Kubernetes Service (AKS) cluster..."

```

```

az aks create --resource-group $RESOURCE_GROUP --name $AKS_NAME
--node-count 1 --enable-addons monitoring --generate-ssh-keys
--attach-acr $ACR_NAME

# Get AKS credentials
echo "Getting AKS credentials..."
az aks get-credentials --resource-group $RESOURCE_GROUP --name $AKS_NAME

# Create Azure SQL Database and Server
echo "Creating Azure SQL Server..."
az sql server create --name $SQL_SERVER_NAME --resource-group
$RESOURCE_GROUP --location $LOCATION --admin-user $SQL_ADMIN_USER
--admin-password $SQL_ADMIN_PASSWORD

echo "Creating Azure SQL Database..."
az sql db create --resource-group $RESOURCE_GROUP --server $SQL_SERVER_NAME
--name $SQL_DATABASE_NAME --service-objective S0

# Enable auditing and threat detection on SQL Database
echo "Enabling auditing and threat detection on SQL Database..."
az sql db audit-policy update --name $SQL_DATABASE_NAME --resource-group
$RESOURCE_GROUP --server $SQL_SERVER_NAME --state Enabled
az sql db threat-policy update --name $SQL_DATABASE_NAME --resource-group
$RESOURCE_GROUP --server $SQL_SERVER_NAME --state Enabled

# Create Azure Key Vault
echo "Creating Azure Key Vault..."
az keyvault create --name $KEY_VAULT_NAME --resource-group $RESOURCE_GROUP
--location $LOCATION --sku standard --enablesoft-delete true
--enable-purge-protection true
az keyvault secret set --vault-name $KEY_VAULT_NAME --name "S

```

```

qlAdminPassword" --value $SQL_ADMIN_PASSWORD

# Log in to ACR
echo "Logging in to ACR..."
az acr login --name $ACR_NAME

# Build and push Docker image (this uses Docker for building)
echo "Building Docker image..."
docker build -t $DOCKER_IMAGE_NAME .

echo "Tagging Docker image..."
docker tag $DOCKER_IMAGE_NAME $ACR_URL/$DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG

echo "Pushing Docker image to ACR..."
docker push $ACR_URL/$DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG

# Create Kubernetes deployment YAML file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dwellingly-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: dwellingly-api
  template:
    metadata:
      labels:
        app: dwellingly-api
    spec:
      containers:
        - name: dwellingly-api
          image: $ACR_URL/$DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG

```

```

    ports:
      - containerPort: 80
EOF

# Create Kubernetes service YAML file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: dwellingly-api
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: dwellingly-api
EOF

# Deploy to AKS
echo "Deploying application to AKS..."
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml

echo "Setup and deployment completed successfully."

```

## Instructions to Run the Script

### 1. Save the Script:

- Save the script as `setup_azure_environment.sh`.

### 2. Make the Script Executable:

- Run the following command to make the script executable:

```
chmod +x setup_azure_environment.sh
```

### **3. Run the Script:**

- Execute the script to set up the Azure environment and deploy the application:

```
./setup_azure_environment.sh
```

## **Explanation**

### **1. Logging into Azure:**

- The script starts by logging into Azure using the Azure CLI.

### **2. Resource Creation:**

- It creates a resource group, Azure Container Registry (ACR), Azure Kubernetes Service (AKS) cluster, Azure SQL Server and Database, and an Azure Key Vault.

### **3. Build and Push Docker Image:**

- It builds a Docker image for the Dwellingly API application, tags it, and pushes it to the ACR.

### **4. Kubernetes Deployment:**

- The script creates deployment and service YAML files for Kubernetes and applies them to deploy the application to the AKS cluster.

This script automates the entire setup and deployment process, ensuring a consistent and reproducible environment for the Dwellingly | AI application.