Visual Studio for the Dwellingly | AI application:

Let's break down the setup of the following components in Visual Studio for the Dwellingly | AI application:

1. **Project Structure:**
    - Backend: ASP.NET Core Web API
    - Frontend: Blazor WebAssembly
    - Middleware/Business Logic: .NET Standard Class Library
    - Data Access: .NET Standard Class Library
2. **DevOps CI/CD using Azure DevOps:**
    - Continuous Integration Pipeline
    - Continuous Deployment Pipeline
3. **User Data using Azure SQL Database:**
    - Entity Framework Core for data access
4. **Azure OpenAI Service for Chatbot Integration:**

Step-by-Step Guide

Step 1: Set Up the Project Structure in Visual Studio

1. **Open Visual Studio 2022**.
2. **Create a New Solution:**
    - Go to `File -> New -> Project`.
    - Select `Blank Solution`.

- Name it `Dwellingly`.
- Choose a location for the solution and click `Create`.

3. **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 2: Configure Each Project

2.1 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<Ap
plicationDbContext> 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=tcp:<your_server>.data
```

```
base.windows.net,1433;Initial Catalog=<your_db>;Persist
Security Info=False;User ID=<your_user>;Password=<your_
password>;MultipleActiveResultSets=False;Encrypt=True;T
rustServerCertificate=False;Connection Timeout=30;"
  },
  "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 se
rvices)
    {
        services.AddDbContext<ApplicationDbContext>(opt
ions =>
            options.UseSqlServer(Configuration.GetConne
ctionString("DefaultConnection")));
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWeb
HostEnvironment 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();
        });
    }
}
```

2.2 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.WebAsse
mbly
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.CreateDefa
ult(args);
        builder.RootComponents.Add<App>("#app");

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

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

2.3 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 ApplicationDbContext _context;

        public PropertyService(ApplicationDbContext con
text)
        {
            _context = context;
        }

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

2.4 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 3: Configure 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).

Step 4: Configure User Data and Azure OpenAI for the Chatbot

1. **Azure SQL Database for User Data**:
   - Configure Entity Framework to use Azure SQL Database as shown in the backend configuration.
2. **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.Authoriza
tion = new AuthenticationHeaderValue("Bearer", _apiKe
y);

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

            var responseContent = await response.Conten
t.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; }
    }
}
```

1. **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 mes
sage" />
    <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 HttpC
lient());
        response = await openAIService.GetChatbotRespon
se(userInput);
    }
}
```

Conclusion

Following these steps, you will set up the project structure, configure dependencies, set up CI/CD in Azure DevOps, and integrate user data and Azure OpenAI for the chatbot in Visual Studio. This comprehensive guide ensures that the Dwellingly | AI application is well-structured, scalable, and integrated with modern DevOps practices and AI capabilities.