Simulating the Azure Setup and Deployment in Visual Studio

Simulating the Azure Setup and Deployment in Visual Studio

To simulate the Azure setup and deployment process in Visual Studio, you can leverage Visual Studio's built-in tools and Azure services. Here's a step-by-step guide on how to set up, build, and deploy the Dwellingly | AI application from Visual Studio:

Prerequisites

1. **Visual Studio 2022** installed with the following workloads:
   - ASP.NET and web development
   - Azure development
2. **Azure CLI** installed.
3. **kubectl** CLI installed.
4. **Docker Desktop** installed.
5. **Azure Subscription**.

Step-by-Step Guide

Step 1: Create the 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: Configure the Project

1. **Install Required NuGet Packages**:
   - Open the NuGet Package Manager Console.
   - 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. **Add the DbContext Class**:
   - Add a new class named `ApplicationDbContext.cs` to your project:

```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<Applic
ationDbContext> options)
        : base(options)
    {
    }

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

3. **Configure the Connection String in `appsettings.json`**:

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

```
tificate=False;Connection Timeout=30;"
}
```

4. **Register the DbContext in `Program.cs`**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionSt
ring("DefaultConnection")));
    services.AddControllers();
}
```

Step 3: Create the Dockerfile

1. **Add a Dockerfile to the project root**:

```
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 /ap
p/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 4: Build and Push the Docker Image to ACR

1. **Open Terminal or Command Prompt**:
   - Log in to Azure:

     ```
     az login
     ```

   - Create a resource group:

     ```
     az group create --name DwellinglyResourceGroup --locati
     on eastus
     ```

   - Create Azure Container Registry (ACR):

     ```
     az acr create --resource-group DwellinglyResourceGroup
     --name DwellinglyACR --sku Basic
     ```

   - Log in to ACR:

     ```
     az acr login --name DwellinglyACR
     ```

   - Build the Docker image:

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

   - Tag the Docker image:

     ```
     docker tag dwellingly-api dwellinglyacr.azurecr.io/dwel
     lingly-api:latest
     ```

   - Push the Docker image to ACR:

```
docker push dwellinglyacr.azurecr.io/dwellingly-api:lat
est
```

Step 5: Create Kubernetes Deployment and Service

1. **Create Deployment and Service YAML files**:

   - Create `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: dwellinglyacr.azurecr.io/dwellingly-api:
latest
        ports:
        - containerPort: 80
```

   - Create `service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: dwellingly-api
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: dwellingly-api
```

Step 6: Deploy to AKS

1. **Open Terminal or Command Prompt**:
   - Create AKS cluster with ACR integration:

     ```
     az aks create --resource-group DwellinglyResourceGroup
     --name DwellinglyAKSCluster --node-count 1 --enable-add
     ons monitoring --generate-ssh-keys --attach-acr Dwellin
     glyACR
     ```

   - Get AKS credentials:

     ```
     az aks get-credentials --resource-group DwellinglyResou
     rceGroup --name DwellinglyAKSCluster
     ```

   - Apply the deployment:

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

   - Apply the service:

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

2. **Verify the deployment**:

```
kubectl get deployments
```

3. **Verify the service**:

```
kubectl get services
```

4. **Get the external IP of the service**:

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

Conclusion

Following these steps will allow you to set up, build, and deploy the Dwellingly | AI application using Azure services directly from Visual Studio. This process ensures a robust and scalable infrastructure for your application, leveraging Azure Kubernetes Service (AKS) and Azure Container Registry (ACR) without relying on Docker for deployment.

Setting Up Dependencies, Environment, Entity Framework, and Docker Image in Visual Studio

To set up dependencies, configure the environment, set up Entity Framework, and create a Docker image in Visual Studio, follow these steps.

Prerequisites

- Visual Studio 2022 installed with the following workloads:
  - ASP.NET and web development
  - Azure development
- Azure CLI installed
- Docker Desktop installed

Step-by-Step Guide

Step 1: Create the 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: Configure the Project

1. **Install Required NuGet Packages**:
    - Open the NuGet Package Manager Console.
    - 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. **Add the DbContext Class**:
    - Add a new class named `ApplicationDbContext.cs` to your project:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<Applic
ationDbContext> 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; }
}
```

3. **Configure the Connection String in `appsettings.json`:**

- Update the `appsettings.json` file with your SQL Server connection string:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:<your_server>.databas
e.windows.net,1433;Initial Catalog=<your_db>;Persist Secur
ity Info=False;User ID=<your_user>;Password=<your_password
>;MultipleActiveResultSets=False;Encrypt=True;TrustServerC
ertificate=False;Connection Timeout=30;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

4. **Register the DbContext in `Program.cs`:**

- Configure the `Program.cs` file to use Entity Framework:

```
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 servi
ces)
    {
        services.AddDbContext<ApplicationDbContext>(option
s =>
            options.UseSqlServer(Configuration.GetConnecti
onString("DefaultConnection")));
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHos
tEnvironment 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();
        });
    }
}
```

5. **Create a Controller for Testing**:
    - Add a new controller named `PropertiesController.cs` to your project:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

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

```
    public PropertiesController(ApplicationDbContext conte
xt)
    {
        _context = context;
    }

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

    [HttpGet("{id}")]
    public async Task<ActionResult<Property>> GetProperty
(int id)
    {
        var property = await _context.Properties.FindAsync
(id);

        if (property == null)
        {
            return NotFound();
        }

        return property;
    }

    [HttpPost]
    public async Task<ActionResult<Property>> PostProperty
(Property property)
    {
        _context.Properties.Add(property);
        await _context.SaveChangesAsync();
```

```
        return CreatedAtAction(nameof(GetProperty), new {
id = property.Id }, property);
    }
}
```

Step 3: Create and Configure Dockerfile

1. **Add a Dockerfile to the project root**:

```
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 /ap
p/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 4: Configure Azure Resources

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

```
az login
```

2. **Create a resource group**:

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

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

```
az sql server create --name dwellingly-sql-server --resour
ce-group DwellinglyResourceGroup --location eastus --admin
-user sqladmin --admin-password YourStrong(!)Password

az sql db create --resource-group DwellinglyResourceGroup
--server dwellingly-sql-server --name DwellinglyDB --servi
ce-objective S0
```

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

```
az acr create --resource-group DwellinglyResourceGroup --n
ame DwellinglyACR --sku Basic
```

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

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

6. **Get AKS credentials**:

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

Step 5: Build and Push the Docker Image to ACR

1. **Log in to ACR**:

```
az acr login --name DwellinglyACR
```

2. **Build the Docker image**:

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

3. **Tag the Docker image**:

```
docker tag dwellingly-api dwellinglyacr.azurecr.io/dwellin
gly-api:latest
```

4. **Push the Docker image to ACR**:

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

Step 6: Create Kubernetes Deployment and Service

1. **Create a Kubernetes deployment YAML file (`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: dwellinglyacr.azurecr.io/dwellingly-api:lat
est
        ports:
        - containerPort: 80
```

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

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

3. **Apply the deployment and service**:

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

4. **Verify the deployment and service**:

```
kubectl get deployments
kubectl get services
```

5. **Get the external IP of the service**

:

```sh
kubectl get service dwellingly-api
```

Conclusion

Following these steps, you will set up dependencies, configure the environment, set up Entity Framework, and create a Docker image in Visual Studio. Additionally, you'll deploy the application to Azure Kubernetes Service (AKS) and Azure Container Registry (ACR). This comprehensive guide ensures a robust and scalable infrastructure for the Dwellingly | AI application.

Setting Up the Project in Visual Studio

Step 1: Create the Solution and Projects

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`.
   - Add the following projects:

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

     Frontend:
     - Select `Blazor WebAssembly App`.

- Name it `Dwellingly.Client`.

Middleware/Business Logic:

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

Data Access:

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

Step 2: Configure Each Project

2.1 Configure Backend (`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;
using Dwellingly.Data.Entities;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<Applic
```

```
ationDbContext> 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 `Startup.cs` to register the `DbContext`:

```
public void ConfigureServices(IServiceCollection servic
es)
{
    services.AddDbContext<ApplicationDbContext>(options
=>
        options.UseSqlServer(Configuration.GetConnectio
nString("DefaultConnection")));
    services.AddControllers();
}
```

2.2 Configure Frontend (`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 Configure Middleware/Business Logic (`Dwellingly.Business`)

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

```
public class PropertyService
{
    private readonly ApplicationDbContext _context;

    public PropertyService(ApplicationDbContext contex
t)
    {
        _context = context;
    }

    public async Task<List<Property>> GetPropertiesAsyn
c()
    {
        return await _context.Properties.ToListAsync();
    }
}
```

2.4 Configure Data Access (`Dwellingly.Data`)

1. **Add Data Entities**:

- Create entities and repositories in this project.
- For example, create a `Property` entity:

```
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:

```
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(string
```

```
prompt)
    {
        var requestContent = new StringContent(JsonConv
ert.SerializeObject(new
        {
            prompt = prompt,
            max_tokens = 150
        }), Encoding.UTF8, "application/json");

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

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

        var responseContent = await response.Content.Re
adAsStringAsync();
        var result = JsonConvert.DeserializeObject<Open
AIResponse>(responseContent);

        return result.Choices.FirstOrDefault()?.Text.Tr
im();
    }
}

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 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 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.