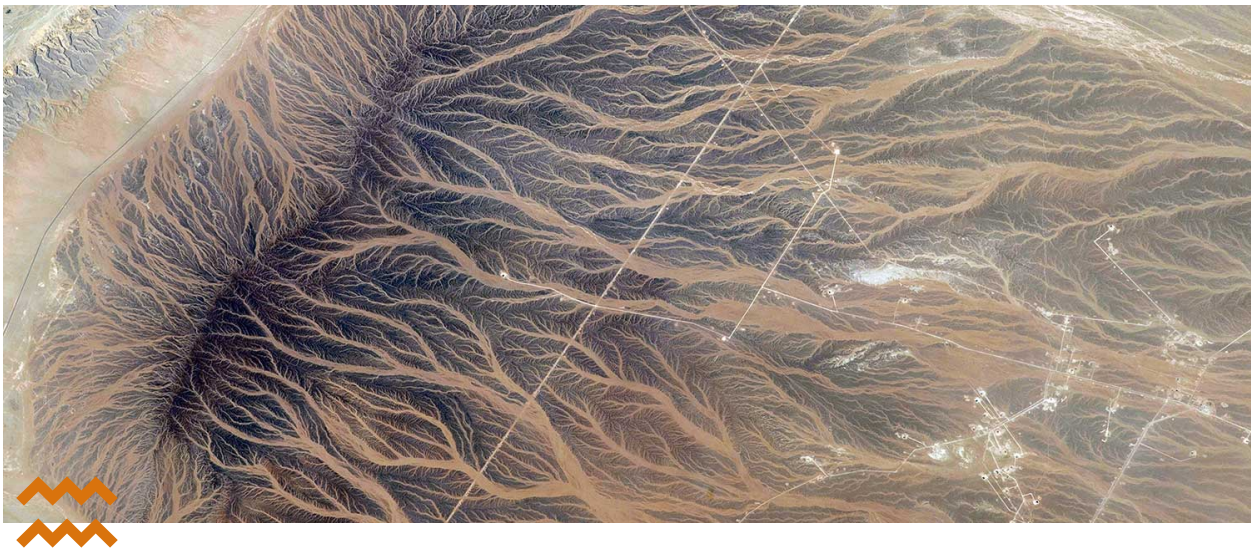

Set up the front end w/ requirements in Visual Studio

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 w
wwroot/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.Component
s.WebAssembly" Version="6.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Component
s.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="@SearchPrope
rtiesAsync">
    <DataAnnotationsValidator />
    <ValidationSummary />
```

```
    <div class="form-group">
        <label for="location">Location:</label>
        <InputText id="location" class="form-control" @bind-V
alue="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</but
ton>
</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">
                <img src="@property.PhotoUrl" alt="@property.
Title" class="property-image" />
                <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 SearchCriteri
a();
    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.SearchProperti
esAsync(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 searchi
ng 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">
        <img src="@property.PhotoUrl" alt="@property.Title" c
lass="property-image" />
        <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: @propert
y.Bathrooms</p>
            <p class="property-description">@property.Descrip
tion</p>
        </div>
    </div>

    <div class="property-actions">
        <button class="btn btn-primary" @onclick="AddToFavori
tesAsync">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(propert
y.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.Cou
nt == 0)
{
    <p>No favorite properties found.</p>
}
else
{
    <div class="favorites-list">
        @foreach (var property in favoriteProperties)
        {
            <div class="favorite-item">
                <img src="@property.PhotoUrl" alt="@property.
Title" class="favorite-image" />
                <div class="favorite-details">
                    <h3 class="favorite-title">@property.Titl
e</h3>
                    <p class="favorite-price">Price: @propert
y.Price.ToString("C")</p>
                    <p class="favorite-location">Location: @p
roperty.Location</p>
                    <div class="favorite-actions">
                        <button class="btn btn-danger" @oncli
ck="() => RemoveFromFavoritesAsync(property.Id)">Remove</butt
on>
                        <a href="/property-details/@property.
Id" class="btn btn-primary">View Details</a>
                    </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.GetFav
oritePropertiesAsync();
        }
        catch (Exception ex)
        {
            errorMessage = "An error occurred while loading f
avorite 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 propertyI
d)
    {
        try
        {
            // Show confirmation dialog before removing from
favorites
            bool confirmed = await JSRuntime.InvokeAsync<bool
>("confirm", "Are you sure you want to remove this property f
rom favorites?");

            if (confirmed)
            {
                await PropertyService.RemoveFromFavoritesAsyn
c(propertyId);
                favoriteProperties = await PropertyService.Ge
tFavoritePropertiesAsync();
            }
        }
        catch (Exception ex)
        {
            // Show error message
            // Log the exception for further investigation
            Logger.LogError(ex, "Error occurred while removin
g 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 c
riteria, 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, ILog
ger<PropertyService> logger)
    {
        _context = context;
        _logger = logger;
    }

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

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

        if (criteria.MinPrice.HasValue)
```

```
        {
            query = query.Where(p => p.Price >= criteria.MinP
rice.Value);
        }

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

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

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

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

    public async Task<Property> GetPropertyByIdAsync(int prop
ertyId)
    {
        return await _context.Properties.FindAsync(propertyI
d);
    }

    public async Task AddToFavoritesAsync(int propertyId)
```

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

    public async Task<List<Property>> GetFavoritePropertiesAs
ync()
    {
        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 propertyI
d)
    {
        var userId = GetUserId();
        var favorite = await _context.Favorites
            .FirstOrDefaultAsync(f => f.PropertyId == propert
yId && 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 service
s)
    {
        services.AddDbContext<ApplicationDbContext>(options =
>
            options.UseSqlServer(Configuration.GetConnectionS
tring("DefaultConnection")));

        services.AddScoped<IPropertyService, PropertyService>
();

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

    public void Configure(IApplicationBuilder app, IWebHostEn
vironment 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(SearchCriteri
a 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, I
Logger<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(c
riteria.Location));
        }

        if (criteria.MinPrice.HasValue)
        {
            query = query.Where(p => p.Price >= criteria.M
inPrice.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>> GetFavoritePropertie
sAsync()
    {
        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 propert
yId)
    {
        var userId = GetUserId();
        var favorite = await _context.Favorites
            .FirstOrDefaultAsync(f => f.PropertyId == prop
ertyId && 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 servi
ces)
    {
        services.AddDbContext<ApplicationDbContext>(option
s =>
            options.UseSqlServer(Configuration.GetConnecti
onString("DefaultConnection")));

        services.AddScoped<IPropertyService, PropertyServi
ce>();

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

    public void Configure(IApplicationBuilder app, IWebHos
tEnvironment 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 kno

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 `ApplicationDbContext.cs`:

```
// ApplicationDbContext.cs

using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<Applicati
onDbContext> 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 ? "dis
abled" : "")">
                <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 == TotalPag
es ? "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="@SearchPr
opertiesAsync">
```

```
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div class="form-group">
        <label for="location">Location:</label>
        <InputText id="location" class="form-control" @bin
d-Value="searchCriteria.Location" />
    </div>
    <div class="form-group">
        <label for="minPrice">Min Price:</label>
        <InputNumber id="minPrice" class="form-control" @b
ind-Value="searchCriteria.MinPrice" />
    </div>
    <div class="form-group">
        <label for="maxPrice">Max Price:</label>
        <InputNumber id="maxPrice" class="form-control" @b
ind-Value="searchCriteria.MaxPrice" />
    </div>
    <div class="form-group">
        <label for="bedrooms">Bedrooms:</label>
        <InputNumber id="bedrooms" class="form-control" @b
ind-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">
                <img src="@property.PhotoUrl" alt="@proper
ty.Title" class="property-image" />
                <div class="property-details">
                    <h3 class="property-title">@property.T
itle</h3>
                    <p class="property-price">Price: @prop
erty.Price.ToString("C")</p>
                    <p class="property-location">Location:
@property.Location</p>
                    <a href="/property-details/@property.I
d" class="btn btn-primary">View Details</a>
                </div>
            </div>
        }
    </div>

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

@code {
    private SearchCriteria searchCriteria = new SearchCrit
eria();
    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.SearchPrope
rtiesAsync(searchCriteria, currentPage);
            properties = result.Properties;
            totalPages = result.TotalPages;
        }
        catch (Exception ex)
        {
            errorMessage = "An error occurred while search
ing for properties. Please try again.";
            Logger.LogError(ex, "Error occurred while sear
ching 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>> _logge
rMock;
    private readonly ApplicationDbContext _context;
    private readonly PropertyService _propertyService;

    public PropertyServiceTests()
    {
        var options = new DbContextOptionsBuilder<Applicat
ionDbContext>()
            .UseInMemoryDatabase(databaseName: "TestDataba
se")
            .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(propertie
s);
        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 : IClassFixtur
e<WebApplicationFactory<Startup>>
{
    private readonly HttpClient _client;

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

    [Fact]
    public async Task PropertySearch_ReturnsResults()
    {
        // Arrange
        var response = await _client.GetAsync("/property-s
earch");

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

        // Assert
        Assert.Contains("Property Search", responseStrin
g);
    }
}
```

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.Compon
ents.WebAssembly" Version="6.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Compon
ents.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"></scrip
t>
</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.