Creating an eCommerce website from scratch with the specified architecture and functionalities is a comprehensive project. Below is a step-by-step guide to help you set it up. This guide will cover folder structure, file creation, coding, and necessary packages.

**Project Structure**

**Recommended Folder Structure**

jboss-cli

Copy

/ECommerceSolution

/CustomerSite (ASP.NET Core MVC)

/Controllers

/Models

/Views

/wwwroot

/ViewComponents

/Pages

/Services

/AdminSite (React)

/src

/components

/services

/pages

/SharedViewModels (Class Library)

/DTOs

/ViewModels

/BackendAPI (ASP.NET Core API)

/Controllers

/Models

/Data

/Services

/Configurations

/Database

/Migrations

/SeedData

**Step 1: Create the Solution and Projects**

1. **Create the Solution**:

bash

Copy

dotnet new sln -n ECommerceSolution

1. **Create Projects**:
   * Customer Site (MVC):

bash

Copy

dotnet new mvc -n CustomerSite

dotnet sln ECommerceSolution.sln add CustomerSite/CustomerSite.csproj

* + Admin Site (React):

bash

Copy

npx create-react-app AdminSite

* + Shared ViewModels:

bash

Copy

dotnet new classlib -n SharedViewModels

dotnet sln ECommerceSolution.sln add SharedViewModels/SharedViewModels.csproj

* + Backend API:

bash

Copy

dotnet new webapi -n BackendAPI

dotnet sln ECommerceSolution.sln add BackendAPI/BackendAPI.csproj

**Step 2: Configure the Backend API**

**2.1 Install Necessary Packages**

Navigate to BackendAPI and install the following packages:

bash

Copy

cd BackendAPI

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

dotnet add package IdentityServer4

dotnet add package Swashbuckle.AspNetCore

**2.2 Create Models**

Create the Product and Category models in BackendAPI/Models.

**Category.cs**:

csharp

Copy

public class Category

{

public int ID { get; set; }

public string Name { get; set; }

}

**Product.cs**:

csharp

Copy

public class Product

{

public int ID { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

public int CategoryID { get; set; }

public Category Category { get; set; }

}

**2.3 Create DbContext**

Create AppDbContext.cs in BackendAPI/Data:

csharp

Copy

using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

public DbSet<Category> Categories { get; set; }

public DbSet<Product> Products { get; set; }

}

**2.4 Configure Services**

In Startup.cs, configure services:

csharp

Copy

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<AppDbContext>(options =>

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

services.AddControllers();

services.AddSwaggerGen();

}

Add Swagger in Configure method:

csharp

Copy

app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "BackendAPI v1"));

**2.5 Seed Data**

In BackendAPI/SeedData, create a class for seeding data:

csharp

Copy

public static class SeedData

{

public static void Initialize(IServiceProvider serviceProvider)

{

using (var context = new AppDbContext(

serviceProvider.GetRequiredService<DbContextOptions<AppDbContext>>()))

{

// Seed data logic here

}

}

}

Invoke SeedData.Initialize in Program.cs.

**Step 3: Create API Endpoints**

In BackendAPI/Controllers, create ProductsController.cs and CategoriesController.cs for CRUD operations.

**ProductsController.cs**:

csharp

Copy

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

[ApiController]

public class ProductsController : ControllerBase

{

private readonly AppDbContext \_context;

public ProductsController(AppDbContext context)

{

\_context = context;

}

// GET: api/products

[HttpGet]

public async Task<ActionResult<IEnumerable<Product>>> GetProducts()

{

return await \_context.Products.Include(p => p.Category).ToListAsync();

}

// Add other CRUD methods...

}

**Step 4: Configure the Customer Site**

**4.1 Install Packages**

In CustomerSite, install:

bash

Copy

dotnet add package Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation

dotnet add package Microsoft.AspNetCore.Http.Connections

**4.2 Set Up MVC**

* Create views for Home, Categories, and Products in the Views folder.
* Use Razor Pages to render data received from the API.

**HomeController.cs**:

csharp

Copy

public class HomeController : Controller

{

private readonly HttpClient \_httpClient;

public HomeController(HttpClient httpClient)

{

\_httpClient = httpClient;

}

public async Task<IActionResult> Index()

{

var categories = await \_httpClient.GetFromJsonAsync<List<Category>>("api/categories");

return View(categories);

}

}

**4.3 Create Razor Pages**

Create Razor pages for viewing products, product details, ratings, and user authentication.

**Step 5: Implement Admin Site**

**5.1 Set Up React**

In AdminSite/src, create components for managing products and categories. Use Axios to call the API.

**ProductManager.js**:

javascript

Copy

import React, { useEffect, useState } from 'react';

import axios from 'axios';

const ProductManager = () => {

const [products, setProducts] = useState([]);

useEffect(() => {

const fetchProducts = async () => {

const result = await axios.get('/api/products');

setProducts(result.data);

};

fetchProducts();

}, []);

return (

<div>

<h1>Manage Products</h1>

{/\* Render products \*/}

</div>

);

};

export default ProductManager;

**Step 6: Shared View Models**

In SharedViewModels, create DTOs and ViewModels that can be shared between the API and MVC applications.

**CategoryDto.cs**:

csharp

Copy

public class CategoryDto

{

public int ID { get; set; }

public string Name { get; set; }

}

**Step 7: Implement Authentication**

Use IdentityServer4 in the Backend API to handle user registration and authentication. Set up endpoints for login/logout and secure the API.

**Step 8: Finalize the Application**

1. **Unit Testing**: Create tests for Controllers and Services in separate test projects.
2. **Documentation**: Use Swagger to document your API.
3. **Deployment**: Consider deploying using AWS services.

**Step 9: Run and Test**

* Run the projects, test API endpoints with tools like Postman, and ensure the MVC and React apps interact correctly with the API.

**Conclusion**

This guide provides a foundational structure to build your eCommerce website. Each step can be expanded with more specifics, but it should serve as a solid starting point. Good luck with your project!