

ePizzaHub Website

GUIDED PROJECT



Building An ePizzaHub Project

Description

Nowadays, an increasing number of people are choosing to order food online rather than cook at home. If you're thinking of starting a food delivery business, there are a few things you'll need to do to be successful. First, you'll need to choose the type of food you want to deliver.

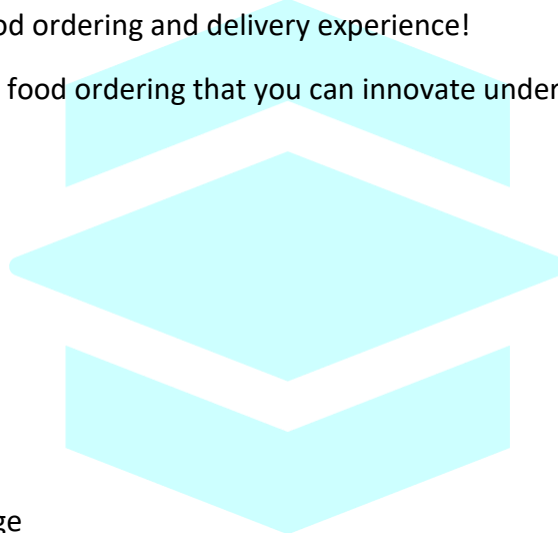
When you've decided on your niche, you'll need to build a website. Your website will be the heart of your business, so it's important to make sure it's user-friendly and attractive. You'll also need to set up a system for taking orders and managing deliveries.

Scope of the Project

We are looking for a website like Domino's, Pizza Hut or McDonald's that enables customers to have a simple, seamless, and effective online food ordering and delivery experience!

Some of the stages of user online food ordering that you can innovate under include but are not limited to:

1. Public Section
 - Home Page
 - About Us Page
 - Contact Us Page
2. Account
 - Login Page
 - SignUp Page
 - Forgot Password Page
 - SignOut Page
3. Items
 - Items Listing Page with Searching and Filtering
 - Add AddToCart Option at Item Level
4. Cart and Payment
 - Cart Page
 - Checkout Page
 - Payment Page with payment gateway integration
 - Payment Confirmation Page
5. User Module
 - Dashboard Page
 - Order Listing Page with Paging and Date Filter



- Order Details Page
- Profile Page
- Change Password Page

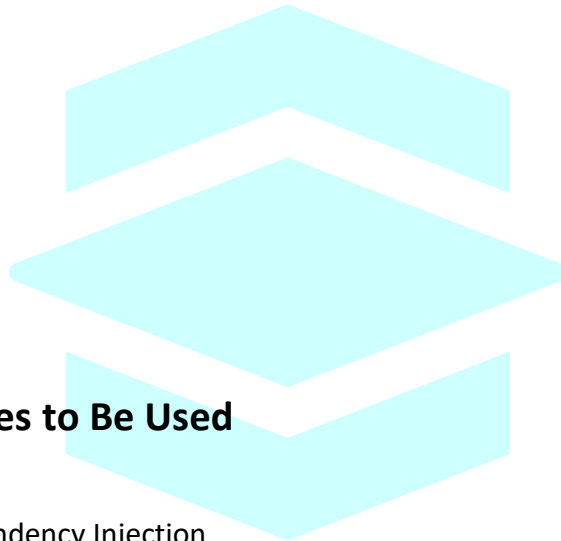
6. Admin Module

- Dashboard Page
- Category Listing, Create and Edit Pages
- Items Listing, Create and Edit Pages
- Received Orders Page
- Profile Page
- Change Password Page

The stages mentioned above are for reference only. While these are some broad highlights, you are encouraged to think out of the box and develop innovative solutions.

Technologies to Be Used

- C# 11
- ASP.NET Core 7
- EF Core 7
- SQL Server
- Bootstrap 5
- jQuery



Architecture and Practices to Be Used

- Clean Architecture
- Repository Pattern, Dependency Injection
- Authentication and Authorization
- Build Mobile Friendly UI
- Bundling and Minification
- Cache
- Errors Logging
- Deploy Code to IIS

Out of the Scope

- Sales Reporting and, Dashboard Capabilities.
- Cancel Order, Track Status of Order
- Delivery Boy Tracking
- Google Map Integration

- Notifications/SMS/Emails
- Payment Cancelled/Refund
- Customer Service/Management.
- OAuth 2.0 authentication via Facebook/Google etc.
- Recommendation Based on Order History
- Supply/Demand Logistics

Prerequisites

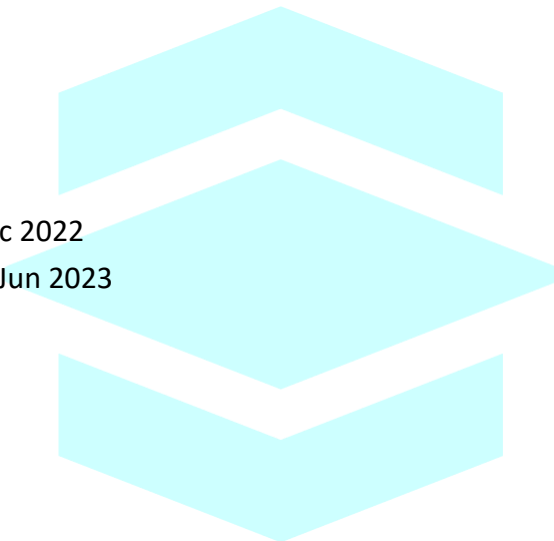
To build this project, you should be familiar with the C#, ASP.NET Core, SQL Server, EF Core and Bootstrap.

Intended Audience

- .NET Beginners
- .NET Developers
- .NET Tech Leads
- .NET Solution Architects

Version History

- Initial Release: 1.0 - 2nd Dec 2022
- Second Release: 2.0 - 26th Jun 2023



Solution: Building an ePizzaHub Project

Requirement Analysis and Planning

Understanding Problem

Nowadays, an increasing number of people are choosing to order food online rather than cook at home. First, you'll need to choose the type of food you want to deliver. Build a website to automate the workflows for taking orders and managing deliveries.

Domain and Subdomains

When building applications, Domain-Driven Design (DDD) refers to the application's problem space as the domains and subdomains. Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

A domain consists of multiple subdomains. Each subdomain corresponds to a different part of the business. Subdomains can be classified using the following criteria:

- **Core:** Most important and key differentiator of an application.
- **Support:** Business-related and used to support business activities.
- **Generic:** Not specific to business but is used to enhance business operations.

For example, the **subdomains** of an **online food delivery domain** application include:

- Authentication Management
- Item Management
- Order Management
- Delivery Management

Ecosystem

The main actors in our Ecosystem are as:

- Customers/Consumers (User)
- Admin (from the restaurant)
- Deliver Boy

Architecture and Database

A well-designed project architecture is essential for any database-driven application. The architecture defines the overall structure of the project components and how they will communicate with each other. It is the heart of an application and decides the UI and database workflows in the system.

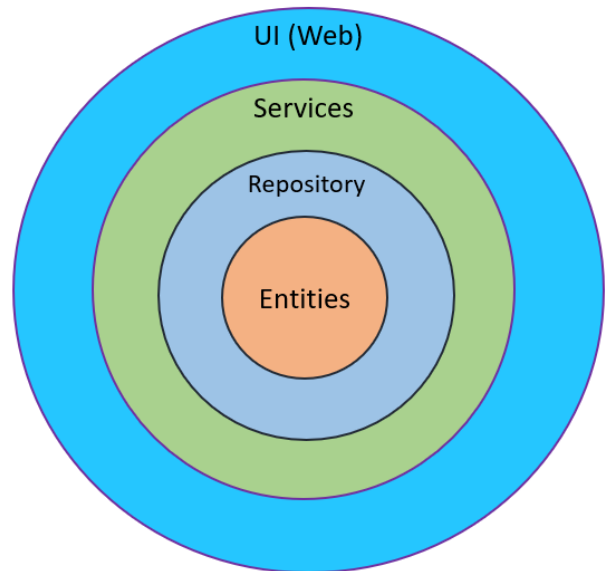
Clean Architecture

The principle behind Clean Architecture is simple: keep your code clean and well organized. This means organizing your code into distinct layers, with each layer serving a specific purpose.

For example, you might have a presentation layer that handles user input and output, a business logic layer that contains the core functionality of your application, and a data access layer that interfaces with your database. By keeping your code organized in this way, you can more easily maintain and update your application as new features are added.

In addition, Clean Architecture makes it easier to unit test your code, as each layer can be tested independently of the others. As a result, Clean Architecture can help you create more reliable and scalable applications.

Onion/Clean Architecture



Technologies Used

- C#
- ASP.NET Core
- EF Core
- SQL Server
- Bootstrap

- jQuery

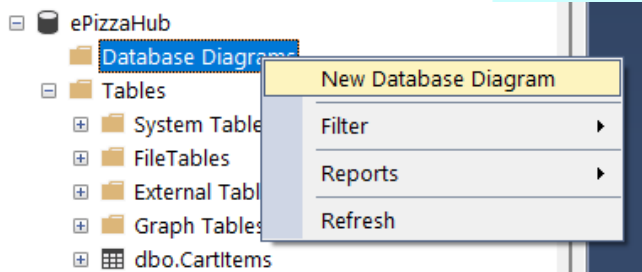
Software and Tools Used

- Visual Studio 2022
- SQL Server Management Studio
- SQL Server
- IIS

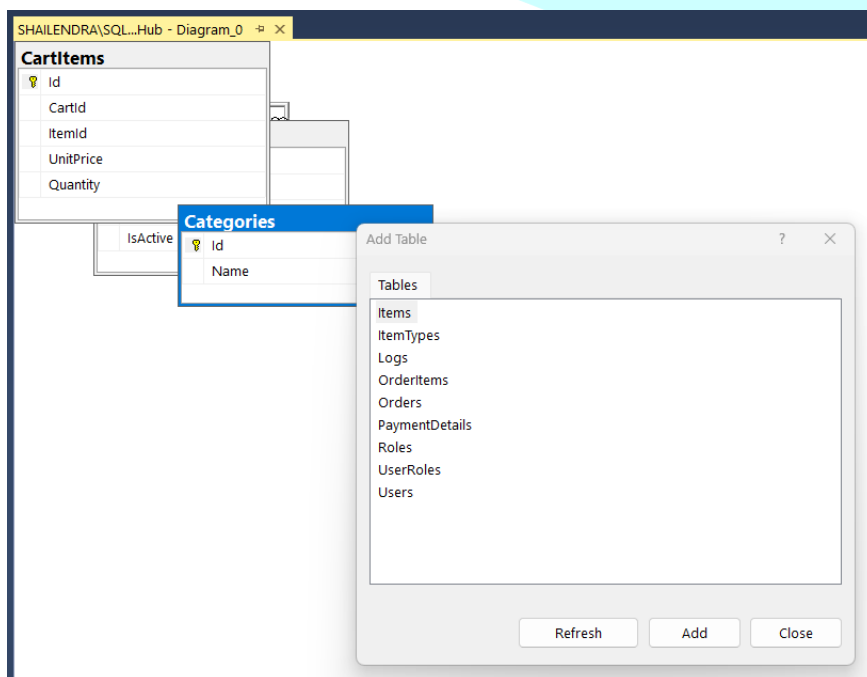
Database Diagram

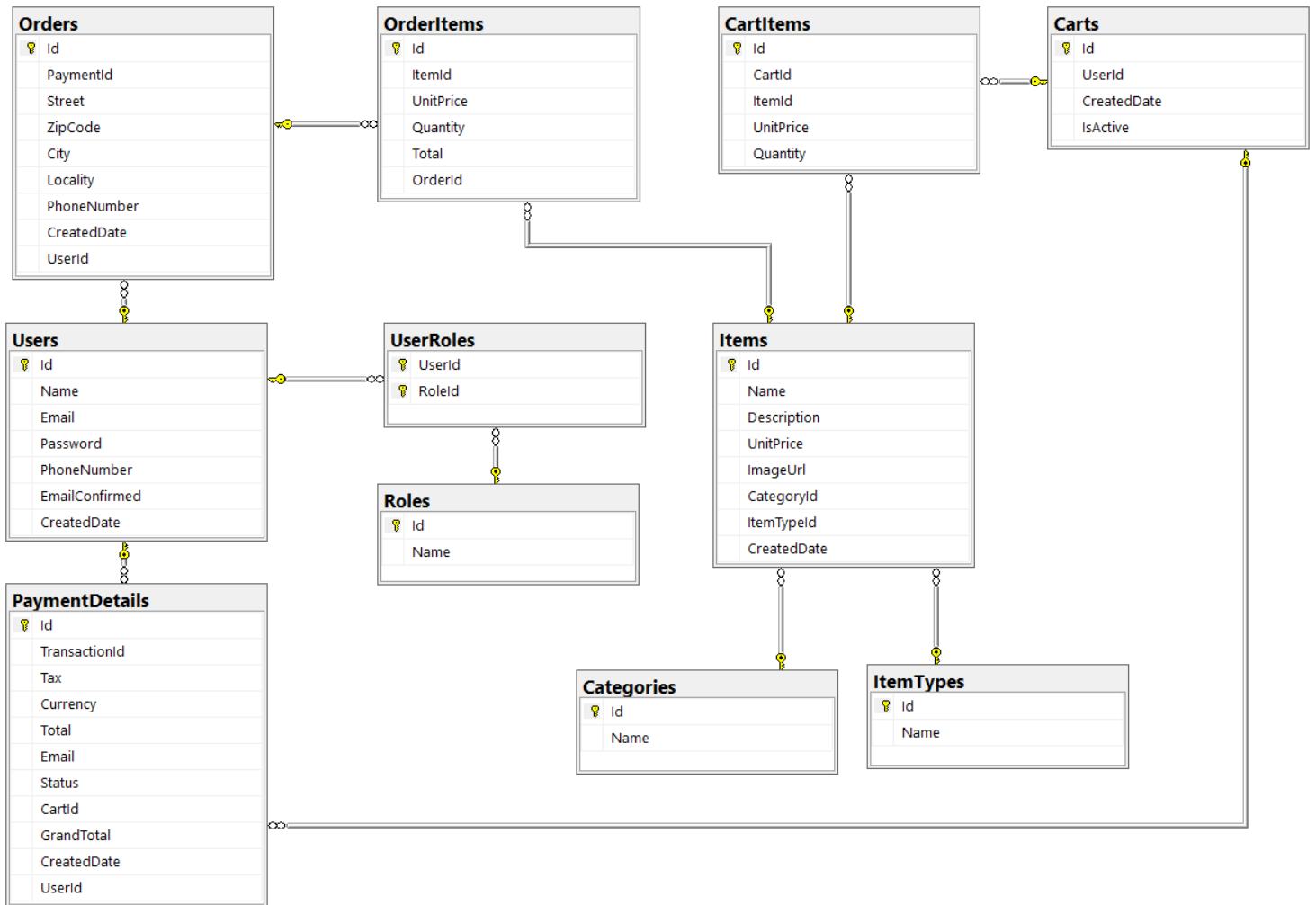
A database diagram is a tool to visually represent the structure of a database. It shows relationships between various entities, such as tables, fields, and keys.

Database diagrams are an essential part of the design process, as they help to ensure that the final product will meet the needs of the users. They also help to avoid problems arising from having an incorrect or incomplete data model. Database diagrams can be created using the SQL Server Management Studio Database Diagram option under a database.



Add tables to include on a database diagram as shown in the pic.



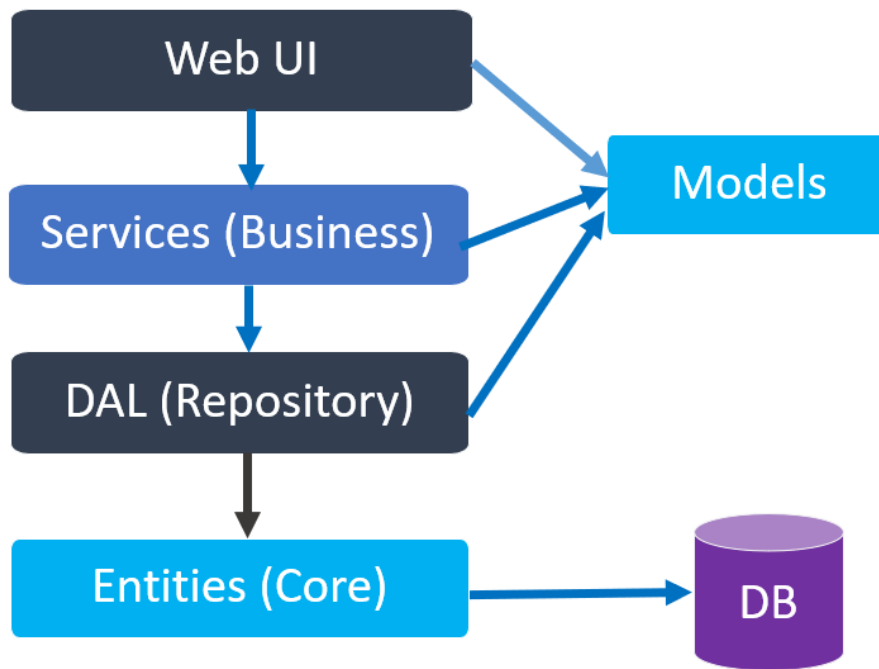


Layers and Patterns Implementation

Creating Layers

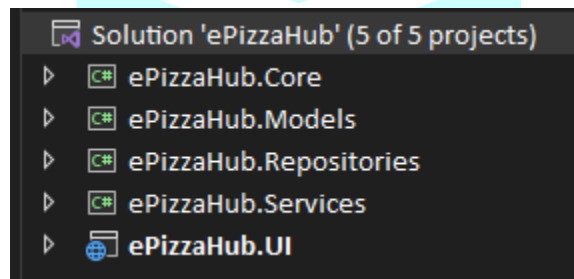
Project layers help to create a well-organized project architecture. By specifying different types of functionalities in separate layers, it becomes easier to reuse code and keep the project maintainable. Furthermore, Project layers can also help to improve performance by reducing dependencies between different parts of the codebase. When creating project layers, it is important to consider the tradeoffs between flexibility and performance.

In general, it is best to start with a small number of layers and then add more as needed. This approach helps to keep the project structure simple and avoid unnecessary complexity.



Create the blank solution in visual studio with the name ePizzaHub then add the following layers:

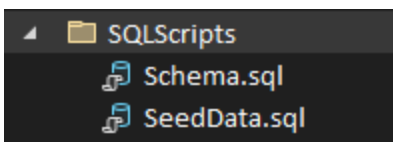
1. ePizzaHub.Core as a project-type class library.
2. ePizzaHub.Models as a project-type class library.
3. ePizzaHub.Repositories as a project-type class library.
4. ePizzaHub.Services as a project-type class library.
5. ePizzaHub.UI as a project-type ASPNetCore Web App (Model-View-Controller).

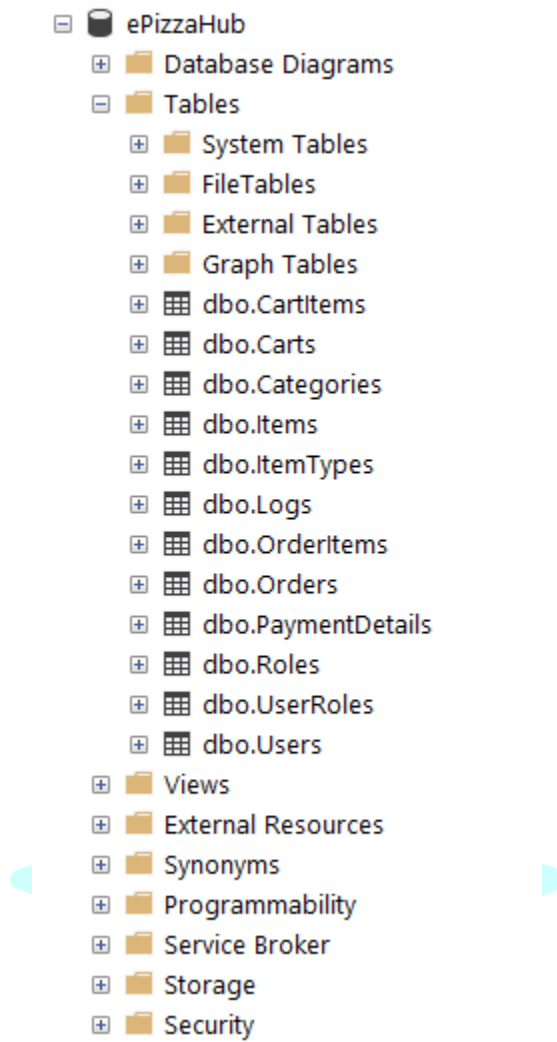


Creating Database

As per the shared database diagrams, you need to create a database and tables. For this one, there are two SQL Scripts files one for Database Schema and another for Seed Data.

Just create a Database First with the name ePizzaHub, then run the shared SQL Schema and Seed Data scripts to create tables and seed master tables with default data for items, users and roles.





The details for Users and Roles are as:

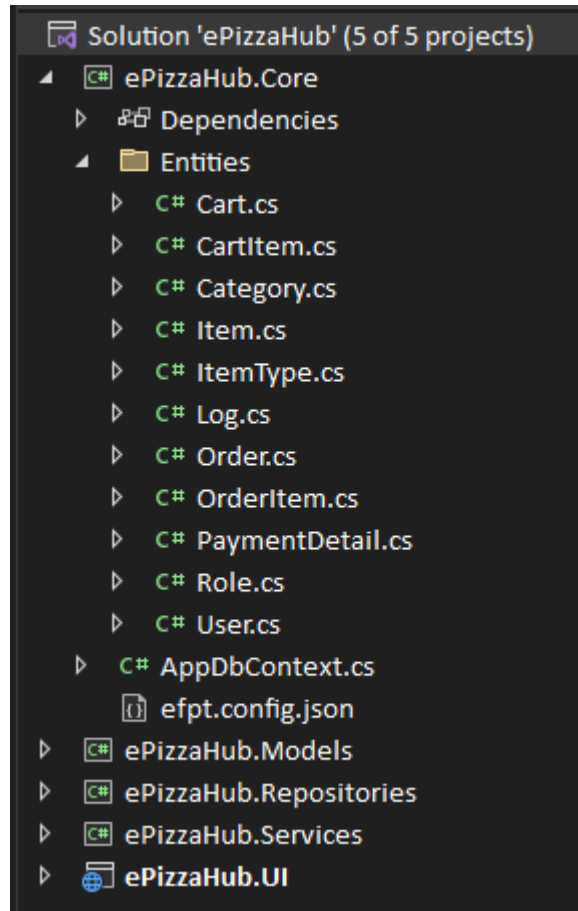
- The default roles are Admin and User. You can create others as well.
- The created user details are as For Admin: admin@gmail.com, Admin@12345678 and For User: user@gmail.com, User@12345678. You can create others as well.

DB First Approach

The Entity Framework Database First Approach allows you to develop your applications using an existing database, which can be very helpful if you already have a database that you need to use for your application.

This approach also allows you to reverse engineer your database into your application to generate or update generated entities. EF Core supports the database first approach using EF Core power tools. The EF Core power tool can be downloaded from here [EF Core Power Tools - Visual Studio Marketplace](#)

The generated code will in the visual studio look like this way.

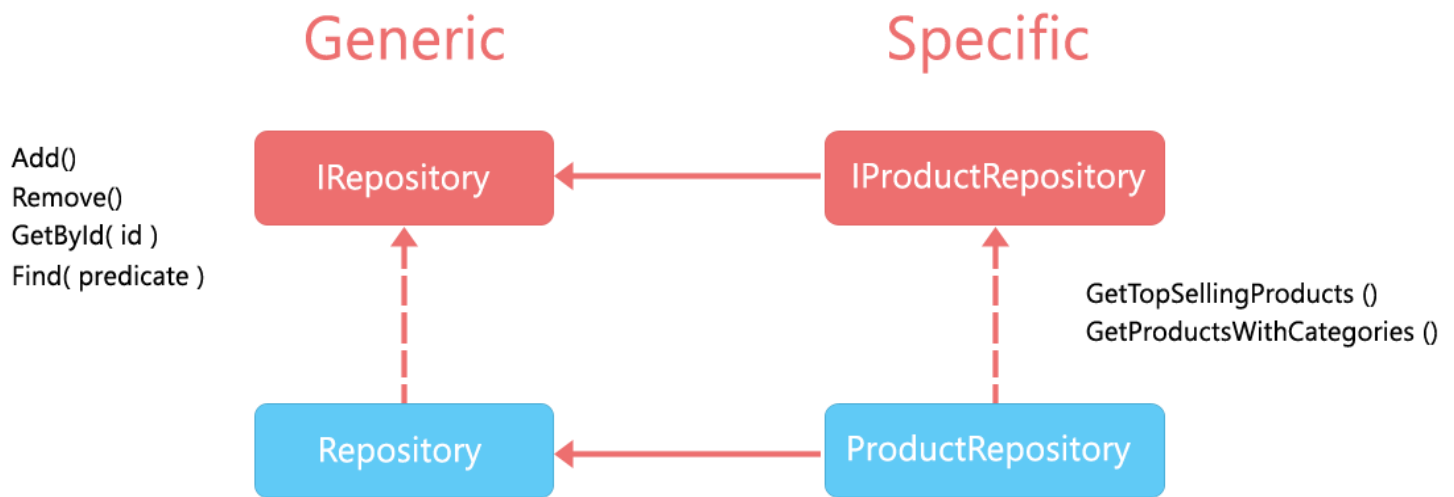


Repository Pattern

The Repository design pattern is a way to help create a separation between the data access layer and the business logic of an application. The pattern is based on the idea of creating a layer where data can be accessed, managed, and manipulated without having to go through the business logic. This can help create more maintainable and testable code.

The Repository pattern can also help to improve performance by caching data and avoiding unnecessary round trips to the database. In addition, the Repository pattern can make it easier to switch between different data sources, such as a relational database and an in-memory cache. Overall, the Repository design pattern can be a valuable tool for creating more robust and scalable applications.

There are two ways to implement repository design patterns- generic and specific. In a generic repository, you need to keep commonly used methods for performing CRUD operations on any entity like Add, Remove, Update, Find, GetById, GetAll etc.



A non-generic (specific) repository implementation is used to define all database operations related to an entity within a separate class. For example, if you have a Product entity then it will have other operations (like GetTopSellingProducts, GetProductsWithCategories and many more) to perform on it.

So better way is, just to create a generic repository for commonly used CRUD operations and for specific ones create a non-generic repository and inherit from a generic repository.

Generic Repository

```
namespace ePizzaHub.Repositories.Interfaces
{
    public interface IRepository<TEntity> where TEntity : class
    {
        IEnumerable<TEntity> GetAll();
        TEntity Find(object Id);
        void Add(TEntity entity);
        void Update(TEntity entity);
        void Delete(object Id);
        int SaveChanges();
    }
}
```

```
using ePizzaHub.Repositories.Interfaces;
using Microsoft.EntityFrameworkCore;

namespace ePizzaHub.Repositories.Implementations
{
```

```

public class Repository<TEntity> : IRepository<TEntity> where TEntity : class
{
    protected DbContext _db;
    public Repository(DbContext db)
    {
        _db = db;
    }
    public void Add(TEntity entity)
    {
        _db.Set<TEntity>().Add(entity);
    }
    public void Delete(object Id)
    {
        TEntity entity = _db.Set<TEntity>().Find(Id);
        if(entity != null)
            _db.Set<TEntity>().Remove(entity);
    }
    public TEntity Find(object Id)
    {
        return _db.Set<TEntity>().Find(Id);
    }
    public IEnumerable<TEntity> GetAll()
    {
        return _db.Set<TEntity>().ToList();
    }
    public int SaveChanges()
    {
        return _db.SaveChanges();
    }
    public void Update(TEntity entity)
    {
        _db.Set<TEntity>().Update(entity);
    }
}
}

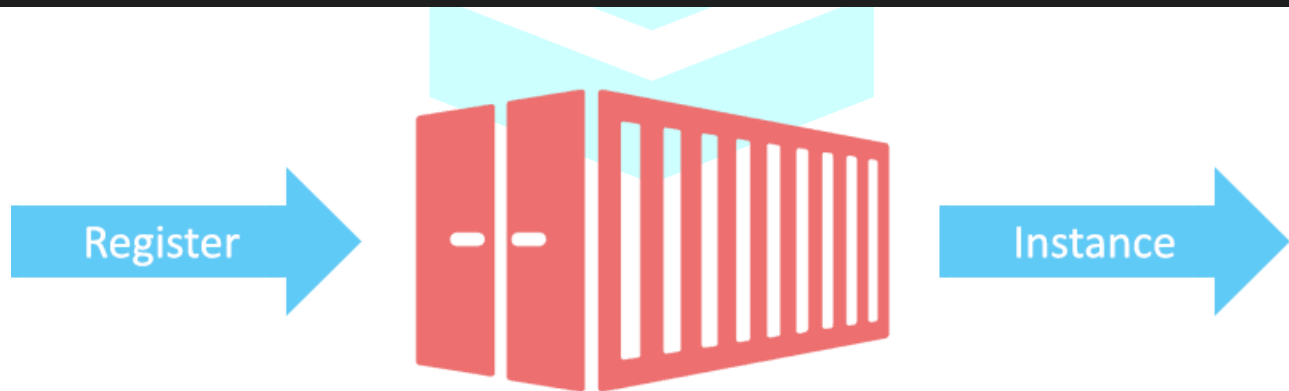
```

Dependency Injection

Dependency injection is a software design pattern which implements IOC. IoC is a programming style where the flow of a program has been inverted i.e. changed from the normal way. It allows the development of loosely coupled software components. In this, components consume functionality defined by the interface without having any knowledge of the class implementation. It helps you to manage future changes and other complexity in software in a better way.

```
using ePizzaHub.Repositories.Interfaces;
using Microsoft.EntityFrameworkCore;

namespace ePizzaHub.Repositories.Implementations
{
    public class Repository<TEntity> : IRepository<TEntity> where TEntity : class
    {
        protected DbContext _db;
        public Repository(DbContext db) //DI: Constructor Injection
        {
            _db = db;
        }
    }
}
```



Manages the lifetime of objects

- **Singleton** - An object of a service is created only once and supplied to all the requests to that service. So, basically, all requests get the same object to work with all calls.
- **Scoped** - An object of a service is created for each request. So, within the scope, it reuses the existing service object.
- **Transient** - An object of a service is created every time when it is requested. Works best for lightweight and stateless services.

Parameter	Singleton	Scoped	Transient
Instance	One for all Requests	One for each request	Every time new
Disposed	App Shutdown	Request End	Request End
Behavior	Stateful & Singleton for all requests	Stateful for a request	Stateless for a request

Creating Class for Dependency Registration

```

using ePizzaHub.Core;
using ePizzaHub.Core.Entities;
using ePizzaHub.Repositories.Interfaces;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using ePizzaHub.Repositories.Implementations;
using ePizzaHub.Services.Interfaces;
using ePizzaHub.Services.Implementations;

namespace ePizzaHub.Services
{
    public class ConfigureService
    {
        public static void RegisterServices(IServiceCollection services,
        IConfiguration configuration)
        {
            //database
            services.AddDbContext<AppDbContext>(options =>
            {
                options.UseSqlServer(configuration.GetConnectionString("DbConnection"));
            });
            services.AddScoped<DbContext, AppDbContext>();

            //repositories: other code removed for clarity
            services.AddScoped<IRepository<Item>, Repository<Item>>();
            services.AddScoped<IUserRepository, UserRepository>();

            //services
            services.AddScoped<ICatalogService, CatalogService>();
        }
    }
}

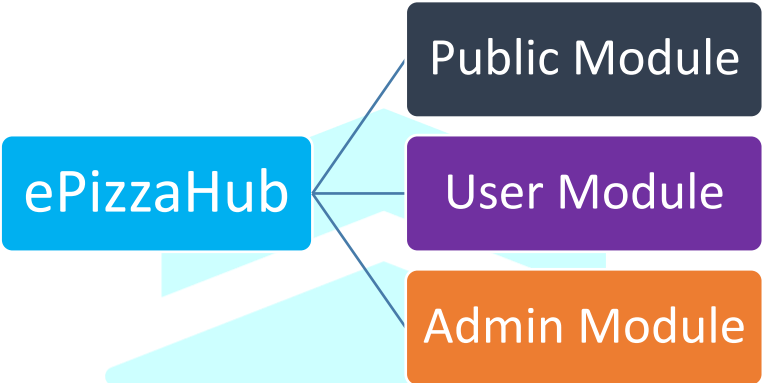
```

```
services.AddScoped<IAuthService, AuthService>();  
}  
}  
}
```

Application Modules Structure

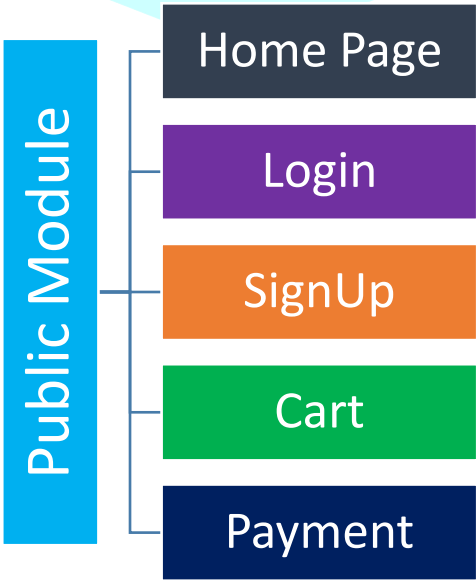
Application code modules allow developers to modularize their code so that it is easier to reuse and manage. Therefore, it is important to decompose an application into code modules to create clean and efficient application code.

The main application is divided into three modules to manage the code files.



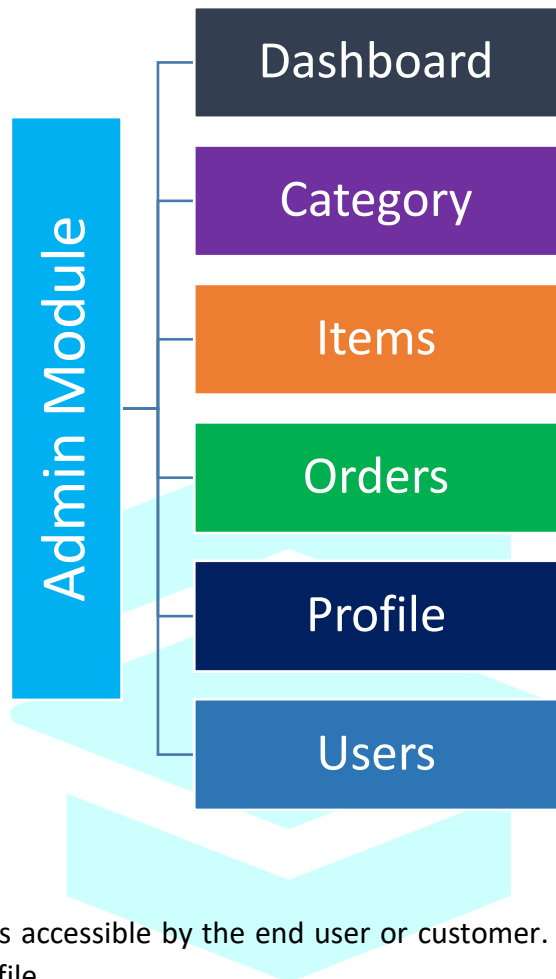
Public Module

A public module is a section that is accessible to the general public. This module can be used by anyone to access the product details and create an account. It will have the following main pages.



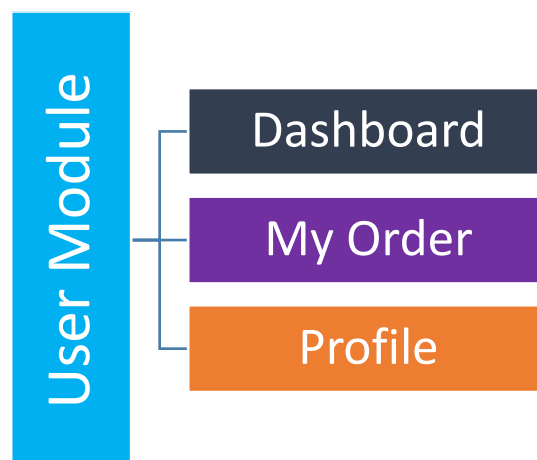
Admin Module

An admin module is a section that is accessible by the company support team and staff. This module will be used to do administrative activities like items management, order management and user management.



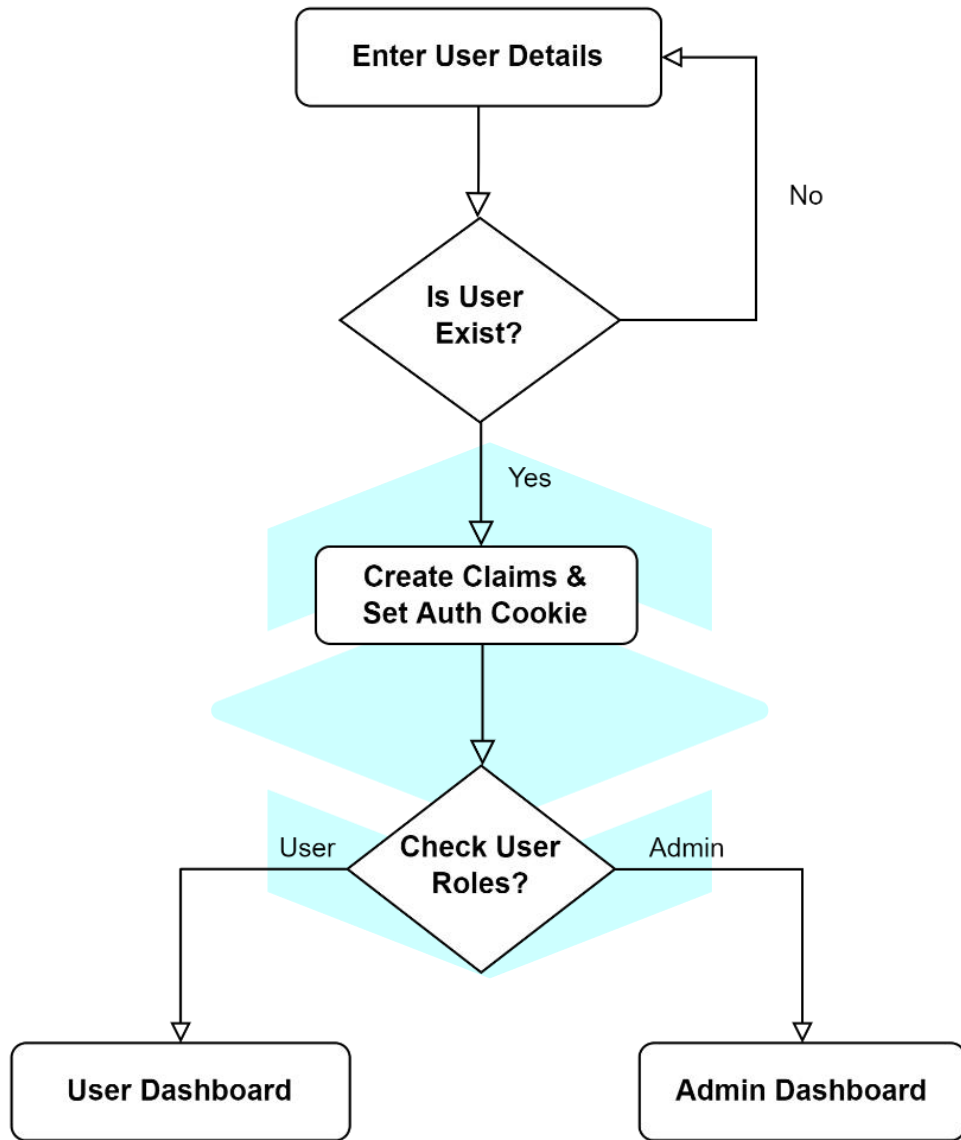
User Module

A user module is a section that is accessible by the end user or customer. This module will be used to check order details and manage his profile.

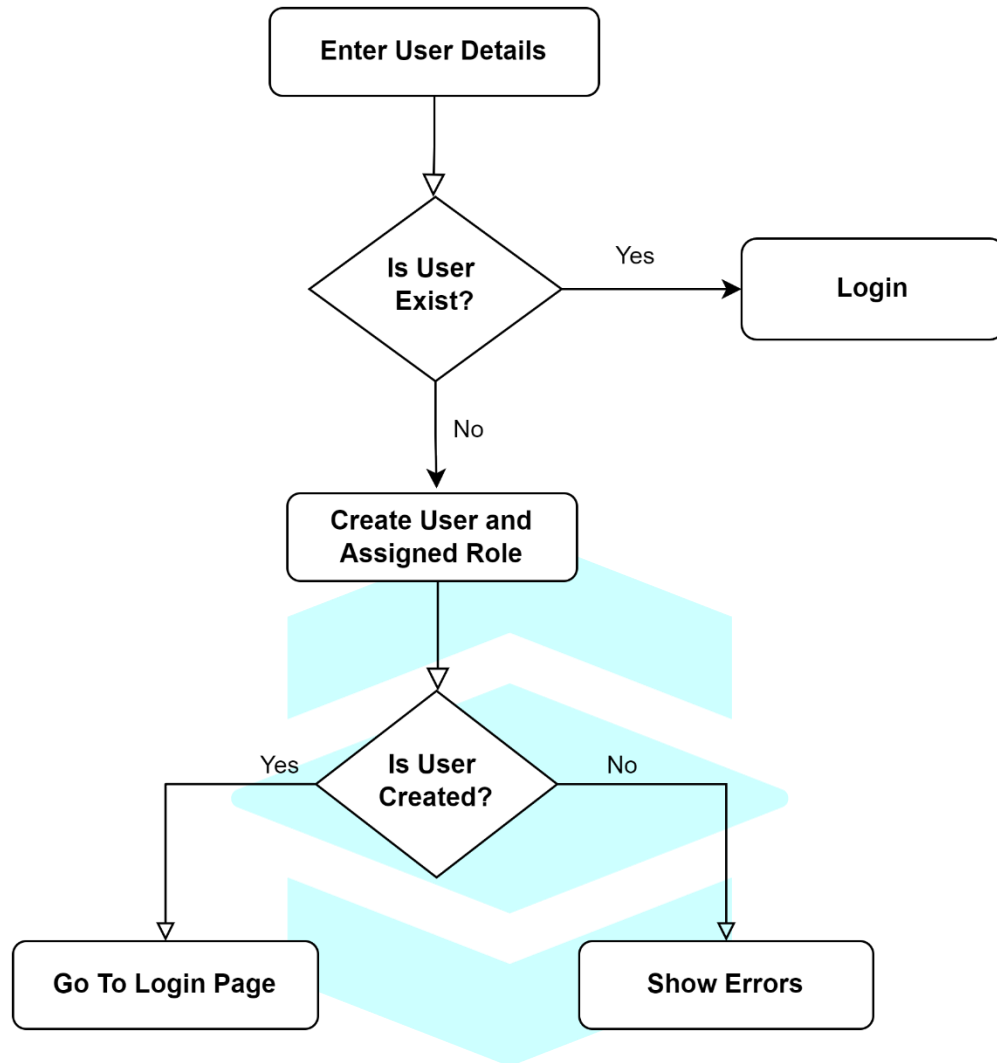


Coding Work Flows

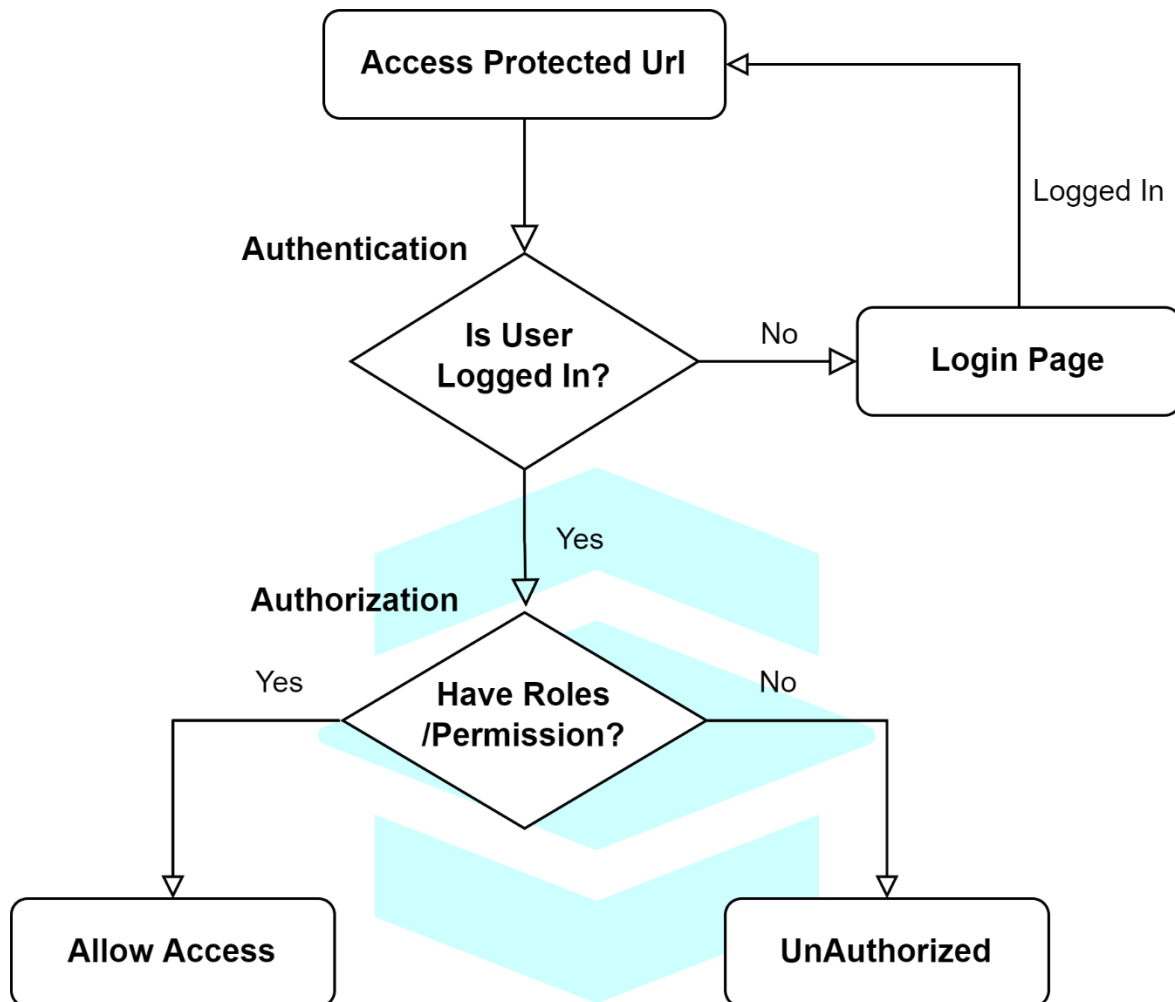
Login Workflow



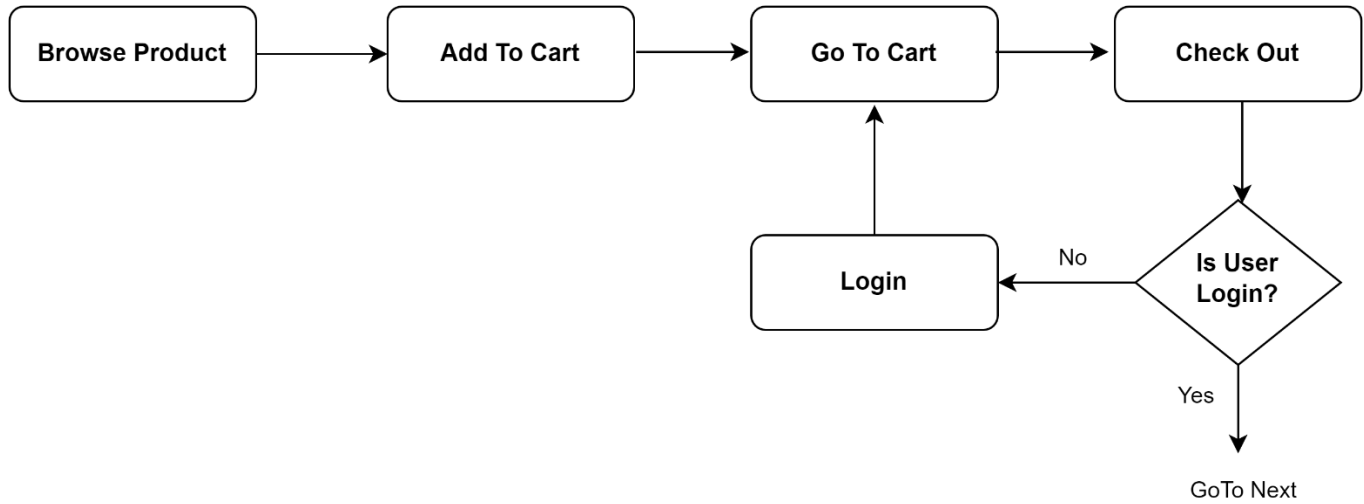
SignUp Workflow



Authentication and Authorization Workflow



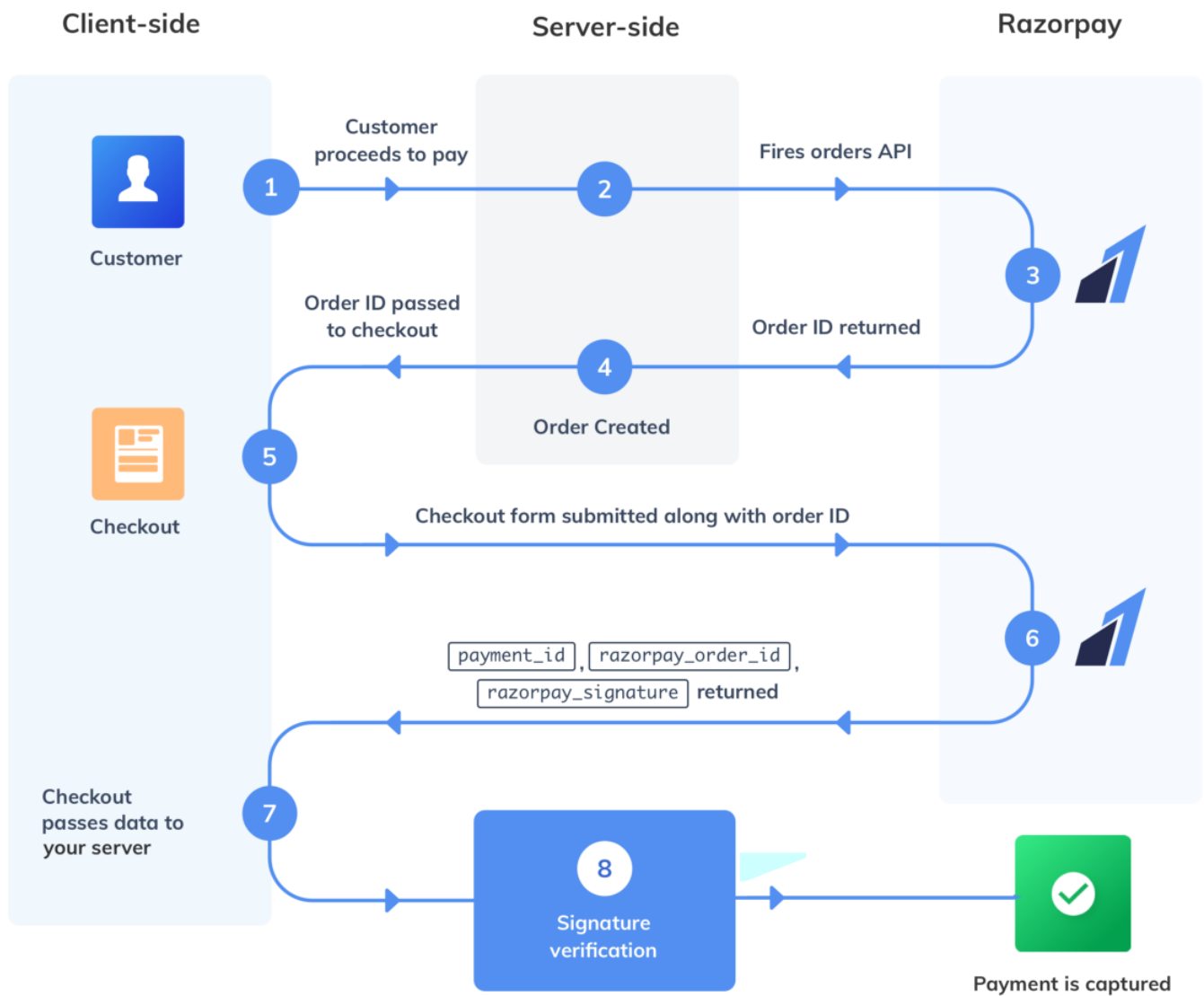
Shopping Cart Workflow

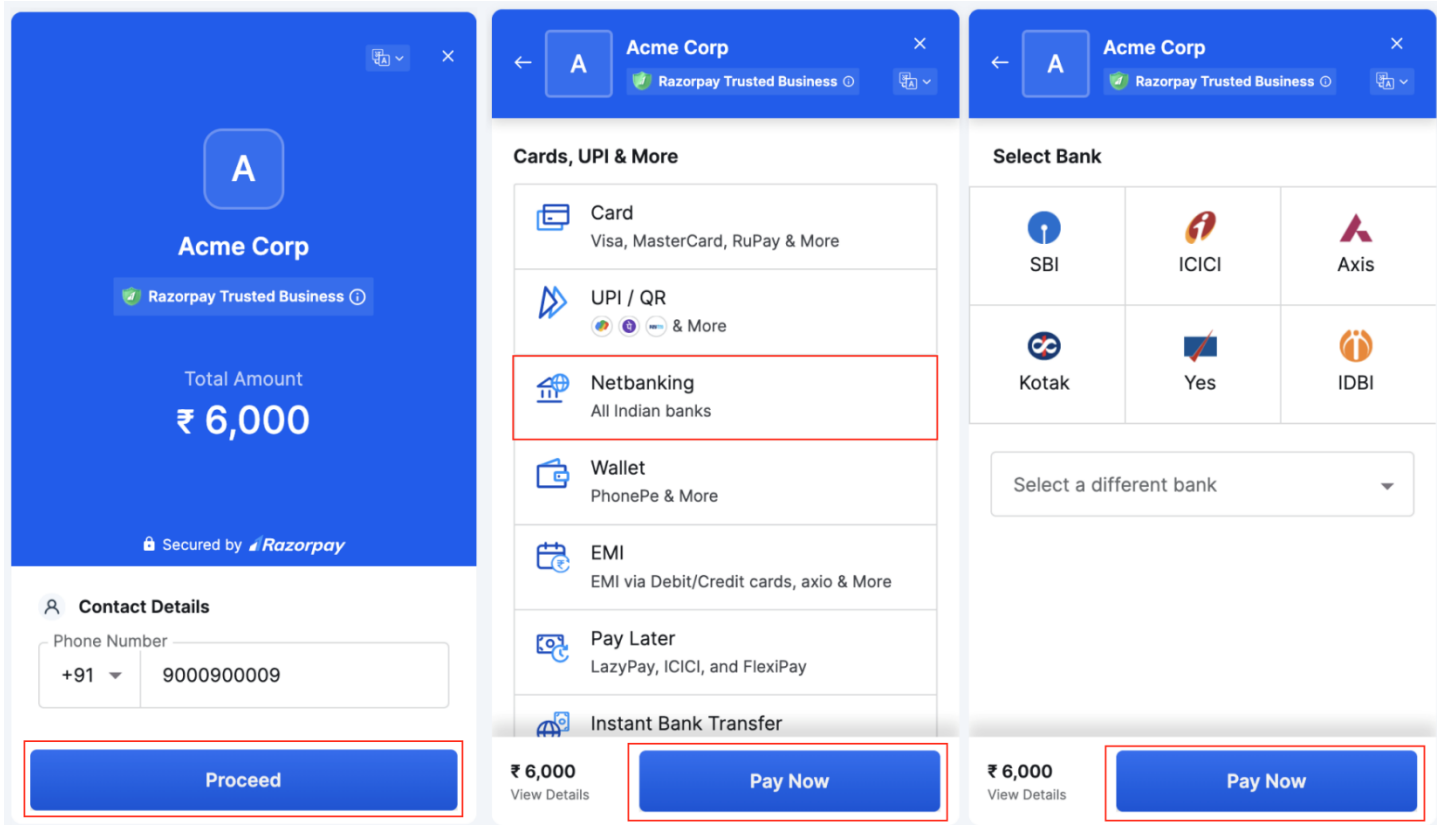


Payment Workflow



Payment Gateway Workflow

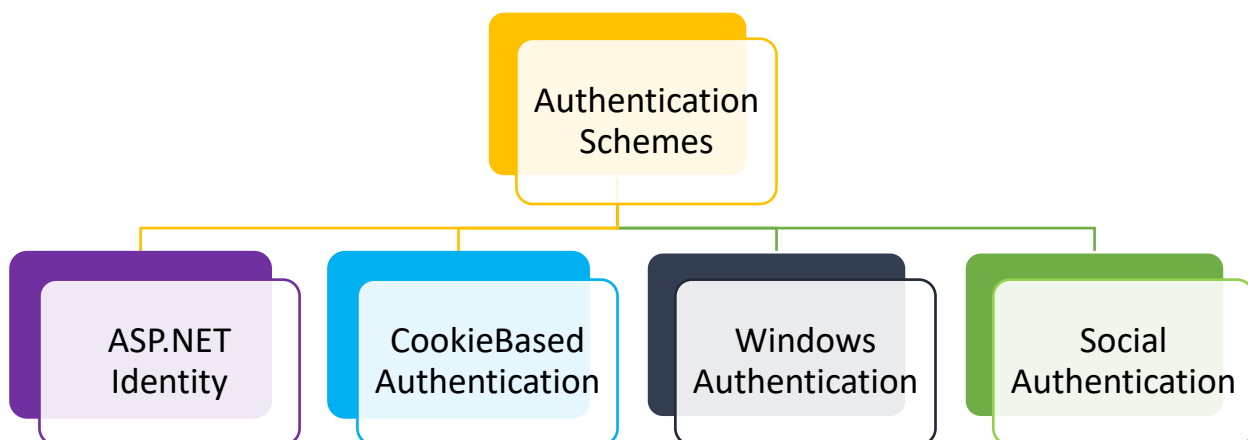




Security and Payment Gateway

Authentication Schemes

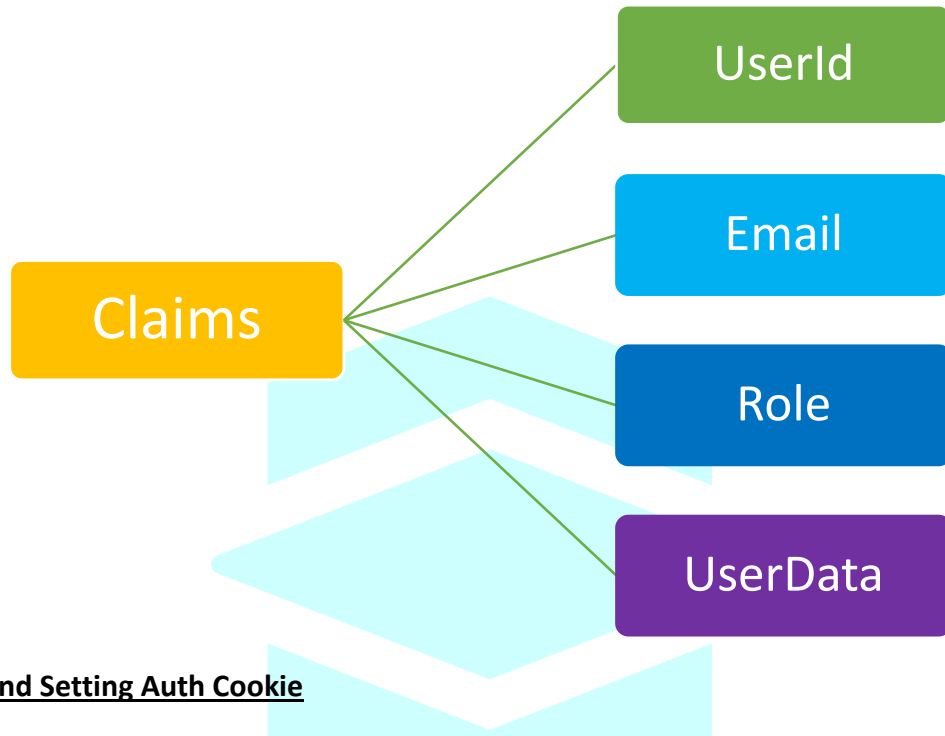
ASP.NET Core is a powerful framework for building web applications. ASP.NET Core supports a variety of authentication schemes, including cookies, Windows-integrated authentication, and third-party login providers such as Google and Facebook. You can also customize the authentication process to meet your specific needs.



Cookie Based Authentication

Cookie base authentication allows you to authenticate users by using an auth cookie. The auth cookie holds user information in the form of claims which you need to set at the time of login.

A claim is a name-value pair that represents the user those details which are unique and help us to authenticate that user and check his permission (authorization). In real life, a person can claim his identity using his Identity Card like (Aadhar Card, Pan Card, Passport etc.)



Creating Claims and Setting Auth Cookie

```
private async void GenerateTicket(UserModel user)
{
    string strData = JsonSerializer.Serialize(user);
    var claims = new List<Claim> {
        new Claim(ClaimTypes.UserData, strData),
        new Claim(ClaimTypes.Email, user.Email),
        new Claim(ClaimTypes.Role, string.Join(",", user.Roles))
    };
    var identity = new ClaimsIdentity(claims,
    CookieAuthenticationDefaults.AuthenticationScheme);
    await
    HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, new
    ClaimsPrincipal(identity), new AuthenticationProperties
    {
```



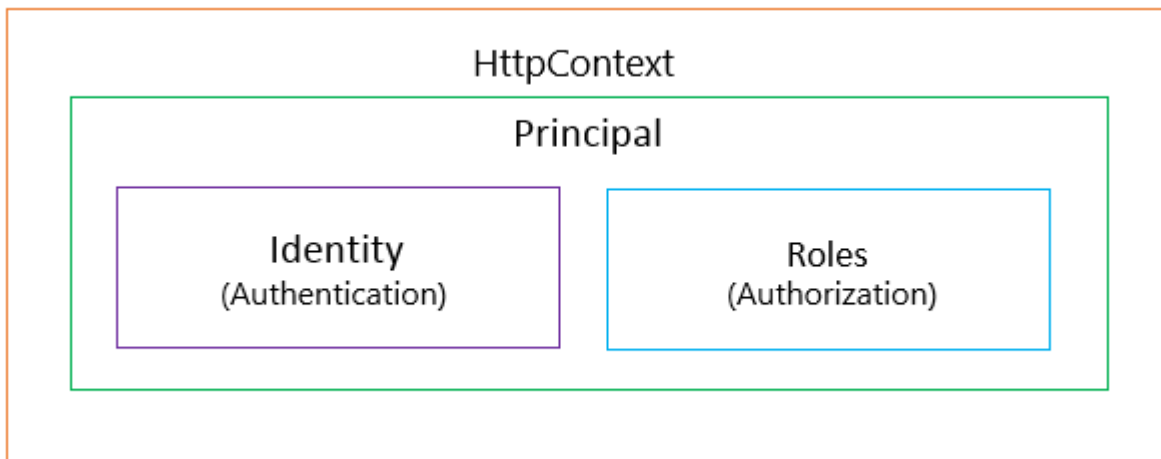
```

        AllowRefresh = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(60),
    });
}

```

Authentication and Authorization

Authentication and Authorization are powerful ways to control access to your web application's protected pages or URLs. By specifying which roles are allowed to access specific resources, you can ensure that only authorized users can access protected pages.



In ASP.NET Core **IAuthorization** filter help you to implement the custom authentication and authorization logic.

```

public class CustomAuthorize : Attribute, IAuthorizationFilter
{
    public string Roles { get; set; }
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        //Check Authentication
        if (context.HttpContext.User.Identity.IsAuthenticated)
        {
            //Check Authorization
            if (!context.HttpContext.User.IsInRole(Roles))
            {
                context.Result = new RedirectToActionResult("Unauthorized", "Account",
new { area = "" });
            }
        }
    }
}

```

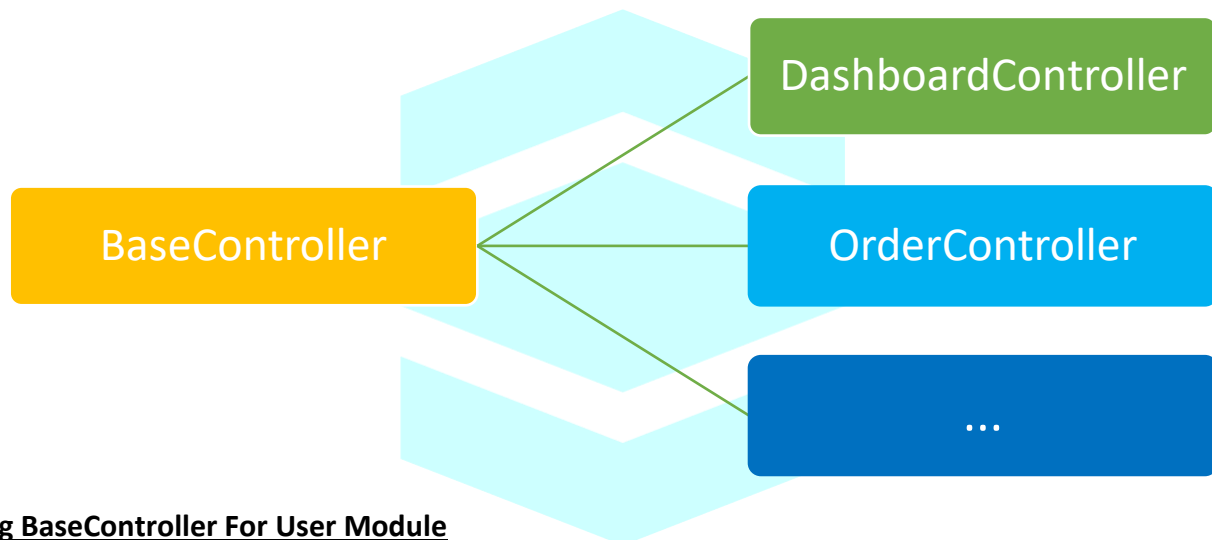
```

        else
        {
            context.Result = new RedirectToActionResult("Login", "Account", new {
area = "" });
        }
    }
}

```

Accessing Logged in User Details

User management is a crucial element of most web applications. There are several ways to access logged-in user details across the application. The best way to access the logged-in user details is to create base classes for controllers and views. Inside the base classes, you can create a public property with the name **CurrentUser** to access the logged-in user details everywhere.



Creating BaseController For User Module

```

[CustomAuthorize(Roles = "User")]
[Area("User")]
public class BaseController : Controller
{
    public UserModel CurrentUser
    {
        get {
            if (User.Claims.Count() > 0)
            {
                string userData = User.Claims.FirstOrDefault(c => c.Type ==
ClaimTypes.UserData).Value;
                var user = JsonConvert.DeserializeObject<UserModel>(userData);
            }
        }
    }
}

```

```

        return user;
    }
    return null;
}
}
}

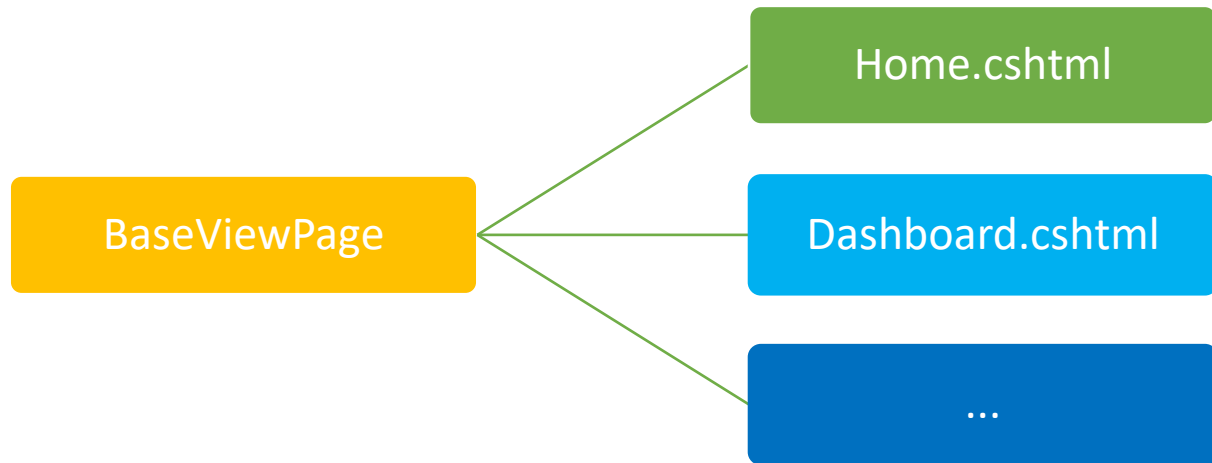
```

Creating BaseController For Admin Module

```

[CustomAuthorize(Roles = "Admin")]
[Area("Admin")]
public class BaseController : Controller
{
    public UserModel CurrentUser
    {
        get {
            if (User.Claims.Count() > 0)
            {
                string userData = User.Claims.FirstOrDefault(c => c.Type ==
ClaimTypes.UserData).Value;
                var user = JsonConvert.DeserializeObject<UserModel>(userData);
                return user;
            }
            return null;
        }
    }
}

```



Creating BaseViewPage

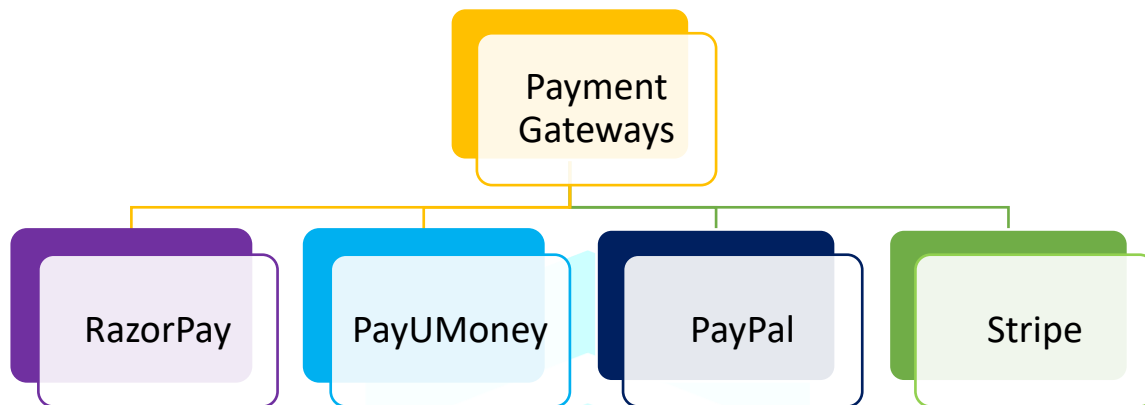
```
public abstract class BaseViewPage<TModel> : RazorPage<TModel>
{
    public UserModel CurrentUser
    {
        get
        {
            if (User.Claims.Count() > 0)
            {
                string userData = User.Claims.FirstOrDefault(c => c.Type ==
ClaimTypes.UserData).Value;
                var user = JsonConvert.DeserializeObject<UserModel>(userData);
                return user;
            }
            return null;
        }
    }
}
```

Adding BaseViewPage Class to ViewImports.cshtml

```
@using ePizzaHub.UI
@using ePizzaHub.UI.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@inherits ePizzaHub.UI.Helpers.BaseViewPage<TModel>
```

Payment Gateway

A payment gateway is a service that allows businesses to accept credit cards and electronic payments. When customers make a purchase, the gateway processes the payment and transmits the funds to the merchant's account. Payment gateways are an essential part of doing business online, as they provide a safe and secure way to accept payments. There are many different payment gateway providers to choose from, and each offers its own unique set of features. When choosing a gateway, it is important to consider your business's needs and find a provider that offers the right mix of features and security.



RazorPay is a payments platform that allows businesses to accept, process, and disburse payments with ease. It offers a wide range of features, including a customizable checkout experience, fraud protection, and support for multiple currencies.

RazorPay has an easy-to-use SDK to integrate into a .NET application. In this project, you will learn the integration of RazorPay. For Razorpay code integration, do refer to the details document [here](#).

Setting RazorPay Checkout Page

```
<script src="https://checkout.razorpay.com/v1/checkout.js"></script>
<script>
    var options = {
        "key": "@Model.RazorpayKey",
        "amount": "@(Model.GrandTotal*100)",
        "currency": "@Model.Currency",
        "name": "@Model.Name",
        "description": "@Model.Description",
        "image": "/images/logo.png",
        "order_id": "@Model.OrderId",
        "handler": function (response){
            $('#rzp_paymentid').val(response.razorpay_payment_id);
        }
    };
    // Razorpay SDK
    var rzp = new Razorpay(options);
    rzp.open();
</script>
```

```

        $('#rzp_orderid').val(response.razorpay_order_id);
        $('#rzp_signature').val(response.razorpay_signature);
        $('#PaymentForm').submit();
    },
    "prefill": {
        "name": "@CurrentUser.Name",
        "email": "@CurrentUser.Email",
        "contact": "@CurrentUser.PhoneNumber"
    },
    "notes": {
        "address": "NA"
    },
    "theme": {
        "color": "#4285F4"
    }
};
var rzp = new Razorpay(options);
window.onload = function(){
    document.getElementById('rzp-button').click();
};
document.getElementById('rzp-button').onclick = function (e) {
    rzp.open();
    e.preventDefault();
};
function submitToPayment() {
    rzp.open();
    e.preventDefault();
}
</script>

```

Consuming Response from RazorPay

```

[HttpPost]
public IActionResult Status(IFormCollection form)
{
    try
    {

```

```

        if (form.Keys.Count > 0 && !string.IsNullOrEmpty(form["rzp_paymentid"]))
        {
            string paymentId = form["rzp_paymentid"];
            string orderId = form["rzp_orderid"];
            string signature = form["rzp_signature"];
            string transactionId = form["Receipt"];
            string currency = form["Currency"];

            var payment = _paymentService.GetPaymentDetails(paymentId);
            bool IsSignVerified = _paymentService.VerifySignature(signature,
orderId, paymentId);

            if (IsSignVerified && payment != null)
            {
                // TO DO
            }
        }
    }
    catch (Exception ex)
    {
    }

    ViewBag.Message = "Your payment has failed. You can contact us at email.";
    return View();
}

```

Extensions to TempData

Extension methods in C# help us to add new methods to an existing class without modifying that class structure. Here, in ASP.NET Core, there is no built-in support for storing/retrieving C# objects and Lists in TempData and Session. So, better to create an extension class for adding the support for storing/ retrieving C# objects and Lists in TempData.

TempData Extension

```

public static class TempDataExtension
{
    public static void Set<T>(this ITempDataDictionary tempData, string key, T value)
    where T : class

```

```

{
    tempData[key] = JsonSerializer.Serialize(value);
}

public static T Get<T>(this ITempDataDictionary tempData, string key) where T : class
{
    tempData.TryGetValue(key, out object o);
    return o == null ? null : JsonSerializer.Deserialize<T>((string)o);
}

public static T Peek<T>(this ITempDataDictionary tempData, string key) where T : class
{
    object o = tempData.Peek(key);
    return o == null ? null : JsonSerializer.Deserialize<T>((string)o);
}
}

```

Best Practices

Data Cache (Server-Side)

The cache is a fast, in-memory caching system that is used to improve the performance of web applications. It stores data in memory so that it can be quickly accessed by the application. The cache can be used to store data that is static or frequently accessed.

For example, in our application, we will cache the list of items to show on our application home page.

In ASP.NET Core you can implement cache using the following three options:

- In-Memory Cache (Recommended for applications running on a single instance)
- Response Cache (Recommended for applications running on a single instance)
- Distributed Cache - Redis (Recommended for applications running on multiple instances)

Response Cache

```

public class HomeController : Controller
{
    [ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
    public IActionResult Index(){ }
}

```


In-Memory Cache

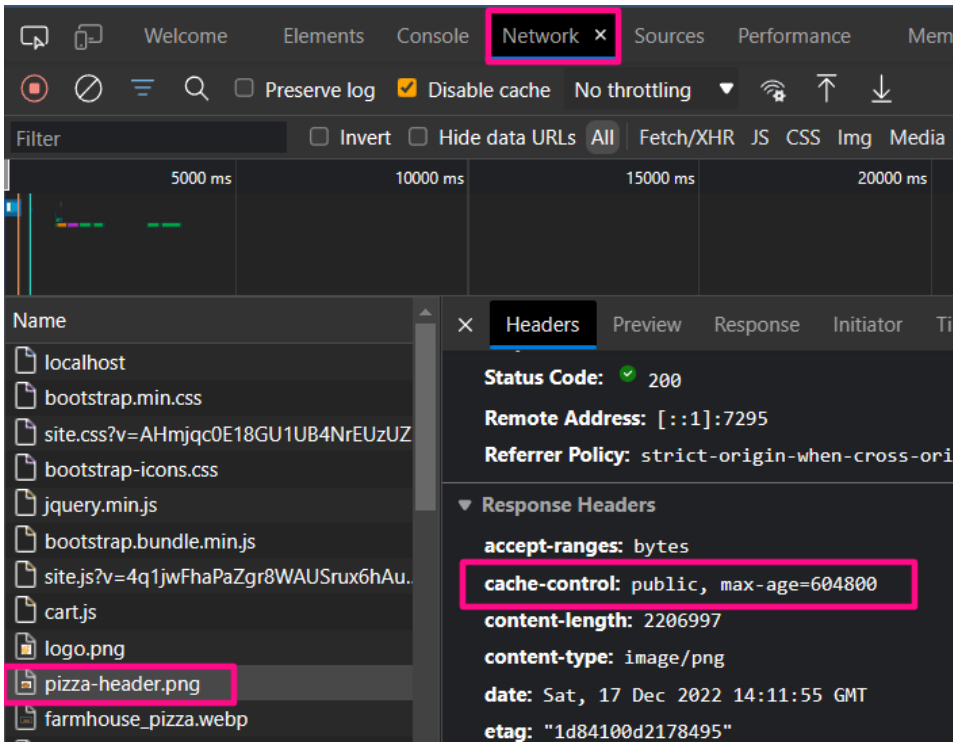
```
public class HomeController : Controller {
    private IMemoryCache _cache;
    public HomeController(IMemoryCache memoryCache) {
        _cache = memoryCache;
    }
    public IActionResult Index() {
        string key = "mykey";
        var cacheEntry = cache.GetOrCreate(key, entry => {
            entry.SlidingExpiration = TimeSpan.FromMinutes(15);
            return GetData();
        });
        return View(cacheEntry);
    }
}
```

Static File Cache (Client-Side)

A static file cache is a type of caching that is used to improve the performance of a website or web application. Static file caches work by storing static files, such as Images, CSS, and JavaScript files, in a cache so that they can be quickly loaded on a web page. This type of caching can help to reduce the amount of time that it takes for a page to load, as well as reduce the bandwidth usage of a site. The Static file caches happen on the client side inside the browser. This cache can provide a significant performance boost for a website or web application.

Setting Up Static File Cache

```
app.UseStaticFiles(new StaticFileOptions
{
    OnPrepareResponse = ctx =>
    {
        const int durationInSeconds = 60 * 60 * 24 * 7; //Secs*Mins*Hrs*Days
        ctx.Context.Response.Headers["cache-control"] =
            "public, max-age=" + durationInSeconds;
    }
});
```



Bundling and Minification

Bundling and minification are two techniques used in asp.net core applications to improve performance. These two techniques can be used together or separately, depending on the needs of the application.

Bundling combines multiple files into a single file, which can reduce the number of HTTP requests made to the server.

Minification removes unnecessary characters from the code, such as whitespace and comments, which can reduce the file size and improve load times.

In general, bundling is most effective when there are a large number of files that are not frequently changed, while minification is most effective for reducing the size of files like CSS and JavaScript files.

To implement it there is one Visual Studio tool is available as given below. You can download it [here](#)



Bundler & Minifier 2022+

Jason Moore |  37,188 installs |  (7) | Free

Adds support for bundling and minifying JavaScript, CSS and HTML files in any project.

[Download](#)

The generated file for bundling and magnification will be like as:

```
[{
  "outputFileName": "wwwroot/css/site.bundle.css",
  "inputFiles": [
    "wwwroot/css/bootstrap.css",
    "wwwroot/css/site.css"]
},
{
  "outputFileName": "wwwroot/js/jquery.bundle.js",
  "inputFiles": [
    "wwwroot/js/jquery.js",
    "wwwroot/js/bootstrap.js",
    "wwwroot/js/jquery.cookie.js"]
}]
```

Environment Specific Rendering

```
<environment names="Development">
  <script src="~/js/cart.js" asp-append-version="true"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Production,Staging">
  <script src="~/js/bundle.min.js" asp-append-version="true"></script>
</environment>
```

Bundled and Minified File

```
<script src="/lib/jquery/dist/jquery.min.js"></script>
<script src="/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/jquery-cookie/1.4.1/jquery.cookie.min.js"></script>
<script src="/js/bundle.min.js?v=O88Rv0UKaE04W9eCEhEHof6J8uI2IYo5j9skwdP6I1I"></script>
```

Error Logging

Error logging is an important part of any application, and ASP.NET Core applications are no exception. There are several different ways to log errors in ASP.NET Core.

Serilog is a .NET logging library that makes it easy to sink logs to a variety of different destinations, including files, databases, and event streams. It also has good support for structured logging, which can be useful for extracting information from log files. In this project, you'll take a look at how to get started with Serilog in an ASP.NET Core application. To set up Serilog you need to download and configure the following NuGet packages:

```
<PackageReference Include="Serilog.AspNetCore" Version="4.1.0" />
<PackageReference Include="Serilog.Enrichers.Environment" Version="2.2.0" />
<PackageReference Include="Serilog.Enrichers.Thread" Version="3.1.0" />
```

```
<PackageReference Include="Serilog.Settings.Configuration" Version="3.2.0" />
<PackageReference Include="Serilog.Sinks.MSSqlServer" Version="5.6.0" />
```

Configure it in program.cs file as:

```
using Serilog;

var builder = WebApplication.CreateBuilder(args);
//logging
builder.Host.UseSerilog((ctx, lc) =>
    lc.ReadFrom.Configuration(ctx.Configuration));
```

Add the following code to appsettings.json to configure Serilog into your project.

```
"Serilog": {
  "Using": [],
  "MinimumLevel": {
    "Default": "Error"
  },
  "WriteTo": [
    {
      "Name": "File",
      "Args": {
        "path": "wwwroot\\Logs\\log.json",
        "formatter": "Serilog.Formatting.Json.JsonFormatter, Serilog"
      }
    },
    {
      "Name": "MSSqlServer",
      "Args": {
        "connectionString": "DbConnection",
        "sinkOptionsSection": {
          "tableName": "Logs",
          "schemaName": "dbo",
          "autoCreateSqlTable": true
        },
        "restrictedToMinimumLevel": "Error"
      }
    }
  ]
}
```

```

    }
  ],
  "Enrich": [
    "FromLogContext",
    "WithMachineName",
    "WithProcessId",
    "WithThreadId"
  ],
  "Properties": {
    "ApplicationName": "Serilog.ePizzaHub"
  }
}

```

HTML Minification

Minifying HTML, CSS and JavaScript files can reduce the size of those files by as much as 50%. This means that your pages will load faster, which is important for both search engine optimization (SEO) and user experience.

WebMarkupMin is a .NET library that can help you minify your HTML, CSS and JavaScript files. It's easy to use and integrates seamlessly with ASP.NET Core. In addition, WebMarkupMin can automatically minify files when they're saved or published, so you don't have to worry about it. As a result, WebMarkupMin can help you improve your website's performance without sacrificing quality or functionality.

First, add WebMarkupMin as a service dependency given below:

```

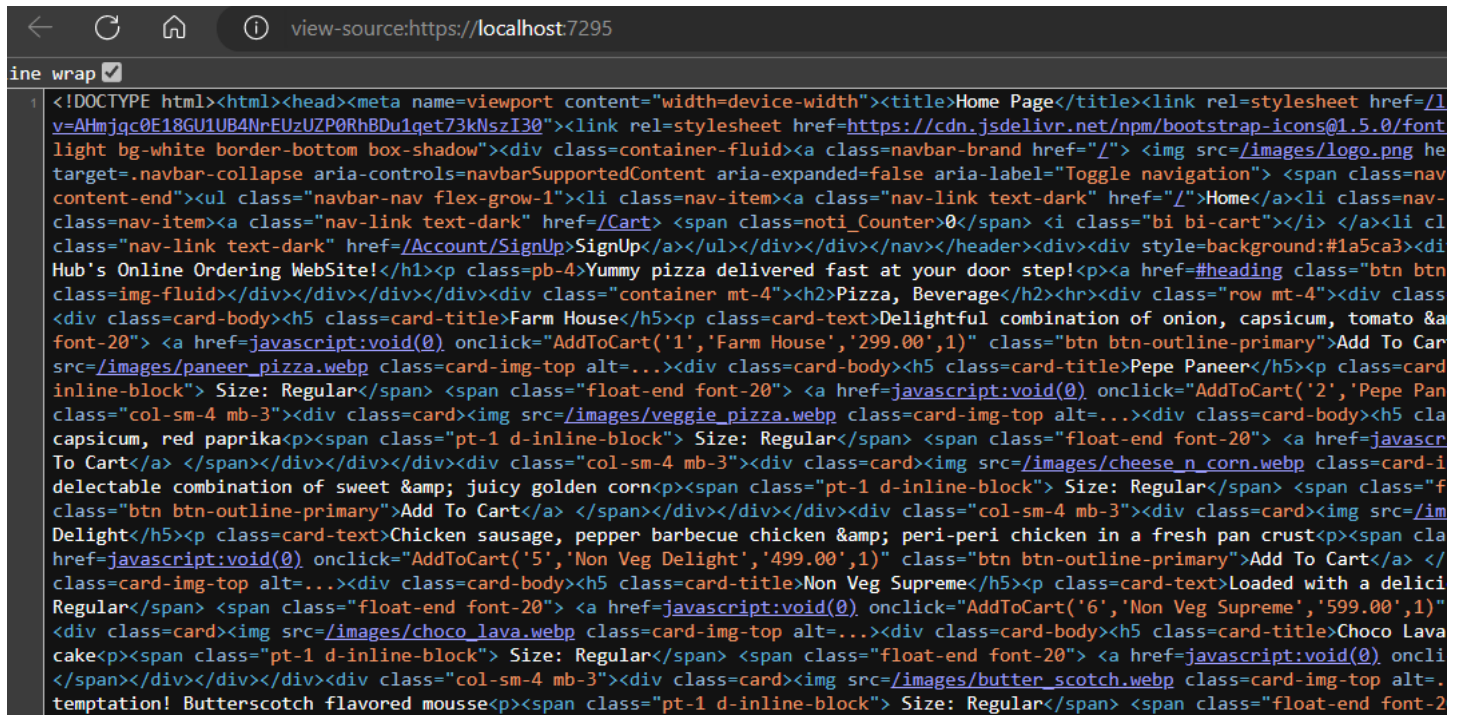
builder.Services.AddWebMarkupMin(options =>
{
    options.AllowMinificationInDevelopmentEnvironment = true;
    options.AllowCompressionInDevelopmentEnvironment = true;
    options.DisablePoweredByHttpHeaders = true;
}).AddHtmlMinification(options =>
{
    options.MinificationSettings.RemoveRedundantAttributes = true;
    options.MinificationSettings.MinifyInlineJsCode = true;
    options.MinificationSettings.MinifyInlineCssCode = true;
    options.MinificationSettings.MinifyEmbeddedJsonData = true;
    options.MinificationSettings.MinifyEmbeddedCssCode = true;
}).AddHttpCompression();

```

Now add, WebMarkupMin in middleware as given below:

```
app.UseRouting();
app.UseWebMarkupMin();
```

HTML Minification



```
<!DOCTYPE html><html><head><meta name=viewport content="width=device-width"><title>Home Page</title><link rel=stylesheet href=/1
v=AHmjgc0E18GU1UB4NrEUzU7P0RhBDu1qet73kNszi30"><link rel=stylesheet href=https://cdn.jsdelivr.net/npm/bootstrap-icons@1.5.0/font
light bg-white border-bottom box-shadow"><div class=container-fluid><a class=navbar-brand href="/"><img src=/images/logo.png he
target=.navbar-collapse aria-controls=navbarSupportedContent aria-expanded=false aria-label="Toggle navigation"><span class=nav
content-end"><ul class="navbar-nav flex-grow-1"><li class=nav-item><a class="nav-link text-dark" href="/">Home</a><li class=nav-
class=nav-item><a class="nav-link text-dark" href=/Cart><span class=noti_Counter>0</span><i class="bi bi-cart"></i></a><li cl
class="nav-link text-dark" href=/Account/SignUp>SignUp</a></ul></div></div></nav></header><div><div style=background:#1a5ca3><di
Hub's Online Ordering WebSite!</h1><p class=pb-4>Yummy pizza delivered fast at your door step!<p><a href=#heading class="btn btn
class=img-fluid"></div></div></div></div><div class=container mt-4><h2>Pizza, Beverage</h2><hr><div class=row mt-4><div class
<div class=card-body><h5 class=card-title>Farm House</h5><p class=card-text>Delightful combination of onion, capsicum, tomato &a
font-20"><a href=javascript:void(0) onclick="AddToCart('1','Farm House','299.00',1)" class="btn btn-outline-primary">Add To Car
src=/images/paneer_pizza.webp class=card-img-top alt=...<div class=card-body><h5 class=card-title>Pepe Paneer</h5><p class=card
inline-block"> Size: Regular</span><span class="float-end font-20"><a href=javascript:void(0) onclick="AddToCart('2','Pepe Pan
class="col-sm-4 mb-3"><div class=card><img src=/images/veggie_pizza.webp class=card-img-top alt=...<div class=card-body><h5 cla
capsicum, red paprika<p><span class="pt-1 d-inline-block"> Size: Regular</span><span class="float-end font-20"><a href=javascr
To Cart</a></span></div></div></div><div class="col-sm-4 mb-3"><div class=card><img src=/images/cheese_n_corn.webp class=card-i
delectable combination of sweet & juicy golden corn<p><span class="pt-1 d-inline-block"> Size: Regular</span><span class="f
class="btn btn-outline-primary">Add To Cart</a></span></div></div></div><div class="col-sm-4 mb-3"><div class=card><img src=/im
Delight</h5><p class=card-text>Chicken sausage, pepper barbecue chicken & peri-peri chicken in a fresh pan crust<p><span cla
href=javascript:void(0) onclick="AddToCart('5','Non Veg Delight','499.00',1)" class="btn btn-outline-primary">Add To Cart</a></
class=card-img-top alt=...<div class=card-body><h5 class=card-title>Non Veg Supreme</h5><p class=card-text>Loaded with a delici
Regular</span><span class="float-end font-20"><a href=javascript:void(0) onclick="AddToCart('6','Non Veg Supreme','599.00',1)"
<div class=card><img src=/images/choco_lava.webp class=card-img-top alt=...<div class=card-body><h5 class=card-title>Choco Lava
cake<p><span class="pt-1 d-inline-block"> Size: Regular</span><span class="float-end font-20"><a href=javascript:void(0) oncli
</span></div></div></div><div class="col-sm-4 mb-3"><div class=card><img src=/images/butter_scootch.webp class=card-img-top alt=.
temptation! Butterscotch flavored mousse<p><span class="pt-1 d-inline-block"> Size: Regular</span><span class="float-end font-2
```

Deployment

Window Server (IIS)

IIS is a web server software that enables you to host your website on a Windows server. Deploying an ASP.NET Core website on IIS requires downloading and setup ASP.NET Core Runtime Hosting Bundle. Once you have everything set up, it's a straightforward process. Here is the download for both of them.

- ASP.NET Core 6 Hosting Bundle: [Download Link](#)
- ASP.NET Core 7 Hosting Bundle: [Download Link](#)

The process of setting the Windows Server or Windows machine is the same for the local environment and Cloud VM for Windows Server running on Azure, AWS or Google Cloud.

Contact Us

You can always reach out to us if you need help with this project. For getting help email us at hello@scholarhat.com or connect to our Discord's **#support** channel.