The C# ASP.NET Core MVC Application Analysis and Walkthrough:

ExpressContactManager

**Startup.cs**

This is the code for the Startup class of an ASP.NET Core web application. The purpose of this class is to configure the web application's services and request handling pipeline.

**Let's go through each part of the code step by step:**

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.Extensions.Logging;

using Microsoft.EntityFrameworkCore;

using ExpressContactManager.Models;

using ExpressContactManager;

using System.Configuration;

namespace ExpressContactManager

{

public class Startup

{

Here, we are importing several namespaces and declaring the Startup class. The Startup class is the entry point for the application and is responsible for configuring services and the request pipeline.

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

In this constructor, an instance of the IConfiguration interface is injected, which is used to access configuration settings from various configuration sources. Here, we're setting the Configuration property of the Startup class to the injected IConfiguration instance.

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

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

// services.AddDbContext<ApplicationDbContext>(options =>

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

services.AddControllersWithViews();

services.AddLogging();

}

In the ConfigureServices method, we are registering the application's services with the dependency injection container. The AddDbContext method is used to register an instance of the ApplicationDbContext class, which is responsible for managing the database context. Here, we're using the UseSqlServer method to configure the ApplicationDbContext to use SQL Server with the connection string defined in the appsettings.json file.

The AddControllersWithViews method is used to register the MVC framework and add support for controllers and views. The AddLogging method is used to add logging to the application.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env, ILogger<Startup> logger)

{

app.UseStaticFiles();

app.UseRouting();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.Use(async (context, next) =>

{

logger.LogInformation($"Request starting {context.Request.Path}");

using (var scope = app.ApplicationServices.CreateScope())

{

var db = scope.ServiceProvider.GetService<ApplicationDbContext>();

await db.Database.EnsureCreatedAsync();

}

await next();

logger.LogInformation($"Request finished {context.Request.Path}");

});

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(name: "default", pattern: "{controller=Contacts}/{action=Index}/{id?}");

});

}

In the Configure method, we are configuring the request pipeline. The UseStaticFiles method is used to serve static files such as images, CSS, and JavaScript files. The UseRouting method is used to route incoming requests to the appropriate controller action.

The if (env.IsDevelopment()) block sets up development-specific error handling, including an exception page that provides detailed error information. The else block sets up production error handling, including redirecting to a predefined error page and enabling HTTPS redirection.

The app.Use method is used to add middleware that logs information about incoming requests. The using statement creates a new scope for the dependency injection container, allowing the ApplicationDbContext to be resolved from the container. The EnsureCreatedAsync method is used to ensure that the database is created and seeded with initial data if necessary.

Finally, the UseEndpoints method is used to define the application's endpoints. Here, we are defining a default controller route that maps to the Index action of the ContactsController. The MapControllerRoute method takes three parameters: name, pattern, and defaults. The name parameter is a unique identifier for the route, the pattern parameter specifies the URL pattern to match, and the defaults parameter specifies default values for the controller, action, and any route parameters.

In summary, this code sets up the configuration for an ASP.NET Core web application, including services, middleware, and routing. It also defines a default controller route and logs information about incoming requests.

**Program.cs**

This is the code for an ASP.NET Core web application. Let's go through each part of the code step by step:

using ExpressContactManager;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

Here, we are importing the ExpressContactManager namespace and the Microsoft.EntityFrameworkCore namespace. We are also creating a new instance of the WebApplication class using the CreateBuilder method and passing in the args array as a parameter.

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

In these lines, we are adding services to the application's dependency injection container using the Services property of the WebApplicationBuilder instance.

The AddControllersWithViews method adds support for controllers and views. The AddDbContext method is used to register an instance of the ApplicationDbContext class, which is responsible for managing the database context. Here, we're using the UseSqlServer method to configure the ApplicationDbContext to use SQL Server with the connection string defined in the appsettings.json file.

var app = builder.Build();

Here, we are building the WebApplication instance and storing it in the app variable.

if (app.Environment.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

In this block of code, we are checking whether the application is running in development mode using the Environment property of the WebApplication instance. If it is running in development mode, we use the UseDeveloperExceptionPage method to display a detailed exception page when an exception occurs. If it is not running in development mode, we use the UseExceptionHandler method to handle exceptions and redirect to an error page.

The UseHsts method is used to set the HTTP Strict Transport Security (HSTS) header, which instructs the browser to only access the website over HTTPS for the specified duration.

app.UseHttpsRedirection();

Here, we are using the UseHttpsRedirection method to redirect all HTTP requests to HTTPS.

app.UseStaticFiles();

The UseStaticFiles method is used to serve static files such as images, CSS, and JavaScript files.

app.UseRouting();

The UseRouting method is used to route incoming requests to the appropriate controller action.

app.UseAuthorization();

The UseAuthorization method is used to add authorization middleware to the request pipeline.

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

Here, we are defining the application's default controller route using the MapControllerRoute method. The name parameter is a unique identifier for the route, the pattern parameter specifies the URL pattern to match, and the defaults parameter specifies default values for the controller, action, and any route parameters. In this case, we are setting the default controller to Home, the default action to Index, and allowing for an optional id parameter.

app.Run();

Finally, we are calling the Run method to start the application's request processing pipeline. This method will block the thread until the application is stopped.

In summary, this code sets up the configuration for an ASP.NET Core web application, including services, middleware, and routing. It also defines a default controller

**ApplicationDbContext.cs**

This is the code for two classes - ContactsController and ApplicationDbContext - that are used in an ASP.NET Core web application. Let's go through each part of the code step by step:

using ExpressContactManager.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

Here, we are importing the ExpressContactManager.Models namespace, which contains the Contact model class, and the Microsoft.AspNetCore.Mvc namespace, which contains the base Controller class. We are also importing the Microsoft.EntityFrameworkCore namespace, which contains the DbContext class.

namespace ExpressContactManager

{

public class ContactsController : Controller

{

private readonly ApplicationDbContext \_context;

public ContactsController(ApplicationDbContext context)

{

\_context = context;

}

}

`

In this class, we are defining a controller that handles requests related to contacts. The ContactsController class inherits from the Controller base class, which provides access to common functionality such as rendering views and handling HTTP requests.

The ApplicationDbContext instance that we registered in the dependency injection container in the Startup class is injected into the constructor using dependency injection. The \_context field is used to store a reference to the injected ApplicationDbContext instance.

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)

{

}

public DbSet<Contact> Contacts { get; set; }

}

**Contact.cs**

This is the code for the Contact model class, which represents a contact entity in an ASP.NET Core web application. Let's go through each part of the code step by step:

using System;

namespace ExpressContactManager.Models

{

public class Contact

{

Here, we are importing the System namespace and declaring the Contact class.

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Address { get; set; }

public string City { get; set; }

public string State { get; set; }

public string Zip { get; set; }

public string PostalCode { get; set; }

public string Country { get; set; }

public string Company { get; set; }

public string Title { get; set; }

public string Notes { get; set; }

public string Fax { get; set; }

public string Mobile { get; set; }

public string WebPage { get; set; }

public DateTime? BirthDay { get; set; }

public string ContactType { get; set; }

}

}

Here, we are defining the properties of the Contact class, which represent the columns in the corresponding database table. Each property has a get and set accessor, which allows the value to be retrieved and updated.

The Id property is an integer that serves as the primary key for the Contact entity. The remaining properties represent various details about the contact, such as their name, email, phone number, address, company, and title.

The BirthDay property is a nullable DateTime object that represents the contact's birthday. The ContactType property is a string that represents the type of contact, such as "Friend" or "Colleague".

In summary, this code defines the properties of the Contact model class, which represents a contact entity in an ASP.NET Core web application. Each property represents a column in the corresponding database table and stores information about the contact.

**HomeController.cs**

This is the code for the HomeController class, which is a controller in an ASP.NET Core web application. Let's go through each part of the code step by step:

using System.Diagnostics;

using Microsoft.AspNetCore.Mvc;

using ExpressContactManager.Models;

namespace ExpressContactManager.Controllers

{

public class HomeController : Controller

{

Here, we are importing the System.Diagnostics namespace, which contains the Activity class, and the Microsoft.AspNetCore.Mvc namespace, which contains the base Controller class. We are also importing the ExpressContactManager.Models namespace, which contains the ErrorViewModel class.

The HomeController class inherits from the Controller base class, which provides access to common functionality such as rendering views and handling HTTP requests.

public IActionResult Index()

{

return View();

}

This method returns a view for the Index action. The View method returns a view that corresponds to the name of the action by default.

public IActionResult Privacy()

{

return View();

}

This method returns a view for the Privacy action.

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public IActionResult Error()

{

return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });

}

This method returns a view for the Error action. The ResponseCache attribute is used to specify caching behavior for the action's response. In this case, we are disabling caching for the response.

The method takes an optional ErrorViewModel object as a parameter, which is used to pass information about the error to the view.

public IActionResult About()

{

return View();

}

**This method returns a view for the About action.**

public IActionResult Contact()

{

return View();

}

**This method returns a view for the Contact action.**

public IActionResult Services()

{

return View();

}

This method returns a view for the Services action.

public IActionResult Portfolio()

{

return View();

}

**This method returns a view for the Portfolio action.**

public IActionResult Product()

{

return View();

}

**This method returns a view for the Product action.**

public IActionResult Products()

{

return View();

}

This method returns a view for the Products action.

In summary, this code defines the HomeController class, which handles requests for various actions in an ASP.NET Core web application. Each method returns a corresponding view for the action.

**ContactsController.cs**

This is the code for the ContactsController class, which is a controller in an ASP.NET Core web application. It handles requests related to the Contact entity.

**Let's go through each part of the code step by step:**

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using ExpressContactManager.Models;

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

namespace ExpressContactManager.Controllers

{

public class ContactsController : Controller

{

private readonly ApplicationDbContext \_context;

public ContactsController(ApplicationDbContext context)

{

\_context = context;

}

Here, we are importing various namespaces and defining the ContactsController class. The controller inherits from the Controller base class and takes an ApplicationDbContext object as a constructor parameter.

public async Task<IActionResult> Index()

{

return View(await \_context.Contacts.ToListAsync());

}

This method returns a view for the Index action. It retrieves a list of contacts from the \_context object and passes it to the View method to generate the view.

public IActionResult Create()

{

return View();

}

**This method returns a view for the Create action.**

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create([Bind("Id,FirstName,LastName,Email,Phone")] Contact contact)

{

if (ModelState.IsValid)

{

\_context.Add(contact);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(contact);

}

This method handles the POST request for the Create action. It takes a Contact object as a parameter, which is created from form data. If the model state is valid, the contact is added to the \_context object and saved to the database. The method then redirects to the Index action.

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

var contact = await \_context.Contacts

.FirstOrDefaultAsync(m => m.Id == id);

if (contact == null)

{

return NotFound();

}

return View(contact);

}

This method returns a view for the Details action. It takes an optional id parameter, which is used to retrieve the contact with the corresponding Id from the \_context object. If the contact is found, the method returns the contact view. Otherwise, it returns a 404 error.

public async Task<IActionResult> Edit(int id, [Bind("Id,FirstName,LastName,Email,Phone")] Contact contact)

{

if (id != contact.Id)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

\_context.Update(contact);

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!ContactExists(contact.Id))

{

return NotFound();

}

else

{

throw;

}

}

return RedirectToAction(nameof(Index));

}

return View(contact);

}

This method handles the POST request for the Edit action. It takes an id parameter and a Contact object as parameters. If the id parameter does not match the Id property of the Contact object, it returns a 404 error.

If the model state is valid, the method updates the contact in the \_context object and saves the changes to the database. If the database operation fails due to concurrency issues, the method checks whether the contact still exists in the database. If not, it returns a 404 error; otherwise, it re-throws the exception.

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

var contact = await \_context.Contacts

.FirstOrDefaultAsync(m => m.Id == id);

if (contact == null)

{

return NotFound();

}

return View(contact);

}

This method returns a view for the Delete action. It takes an optional id parameter, which is used to retrieve the contact with the corresponding Id from the \_context object. If the contact is found, the method returns the contact view. Otherwise, it returns a 404 error.

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var contact = await \_context.Contacts.FindAsync(id);

\_context.Contacts.Remove(contact);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

}

}

This method handles the POST request for the Delete action. It takes an id parameter, which is used to retrieve the contact with the corresponding Id from the \_context object. The contact is then removed from the \_context object and saved to the database. The method then redirects to the Index action.

Overall, the ContactsController class provides an interface for CRUD operations on the Contact entity. It takes care of interacting with the database through the ApplicationDbContext object and returns views that display the relevant information.

**LaunchSettings.json**

{

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://localhost:32740",

"sslPort": 44386

}

},

"profiles": {

"ExpressContactManager": {

"commandName": "Project",

"dotnetRunMessages": true,

"launchBrowser": true,

"applicationUrl": "https://localhost:5001;http://localhost:5000",

"launchUrl": "https://localhost:5001/Contacts",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": true,

"launchUrl": "https://localhost:44386/Contacts",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

}

**Appsettings.json**

This is a JSON configuration file that contains settings for an ASP.NET Core application.

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContactManagerDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

This section specifies the connection string for the database that the application will use. In this case, the connection string is named DefaultConnection and is set to a local SQL Server Express instance ((localdb)\\mssqllocaldb), using a database named ContactManagerDb. The Trusted\_Connection parameter specifies that the connection should use Windows authentication and the MultipleActiveResultSets parameter enables multiple active result sets.

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

This section specifies the logging settings for the application. The LogLevel key has two nested keys: Default and Microsoft.AspNetCore. The Default level is set to Information, which means that all logs with a level of Information or higher will be included in the output. The Microsoft.AspNetCore level is set to Warning, which means that logs with a level of Warning or higher from the Microsoft.AspNetCore namespace will be included in the output.

"AllowedHosts": "\*"

This section specifies the allowed hosts for the application. In this case, the value is set to \*, which means that any host is allowed to access the application.

Overall, this configuration file specifies the database connection string, logging settings, and allowed hosts for the application.

**Create.cshtml**

This is a Razor view file that allows users to create a new contact by filling out a form.

@model Contact

@{

ViewData["Title"] = "Create";

}

<h2>Create New Contact</h2>

<form asp-action="Create">

This section defines the model for the view (@model Contact) and sets the page title (ViewData["Title"] = "Create"). The form element specifies that the form data should be submitted to the Create action of the Contacts controller.

<div class="form-group">

<label asp-for"FirstName">First Name</label>

<input asp-for="FirstName" class="form-control" />

</div>

This section creates a form group for the first name field. The asp-for attribute is used to bind the input to the corresponding property of the Contact model. The class="form-control" attribute is used to apply Bootstrap styling to the input.

<button type="submit" class="btn btn-primary">Create</button>

</form>

This section adds a button to submit the form.

**Here are what each piece does of <@razor\_html>.cshtml files do in greater detail:**

This is a Razor view file that creates a form to add a new contact to the contact manager application.

@model

Contact declares the model used by the view as Contact. The view is strongly-typed and expects

a Contact object to be passed to it.

@{ ViewData["Title"] = "Create"; }

Sets+ the title of the page to "Create".

<h2>Create New Contact</h2>

creates a heading for the form to indicate that it is used to create a new contact.

<form asp-action="Create">

creates an HTML form that will be submitted to the Create action method in the controller.

<div class="form-group">

creates a new form group that can contain a label and an input.

<label asp-for"FirstName">First Name</label>

creates a label for the first name input. The asp-for attribute is used to bind the label to the corresponding property of the Contact model.

<input asp-for="FirstName" class="form-control" />

creates an input field for the first name. The asp-for attribute is used to bind the input to the corresponding property of the Contact model. The class="form-control" attribute is used to apply Bootstrap styling to the input.

The next several div elements, labels, and inputs follow the same pattern as the first name input, allowing the user to enter their last name, email, phone number, address, city, state, zip code, notes, and contact type.

<button type="submit" class="btn btn-primary">Create</button>

creates a button to submit the form. The type="submit" attribute specifies that this button should submit the form when clicked. The class="btn btn-primary" attribute applies Bootstrap styling to the button.

This Razor view file is designed to display a form that allows users to create a new contact. The first line of the code (@model Contact) specifies that the view uses the Contact model to display data.

The second block of code (@{...}) defines the title of the view (Create) and sets up ViewData, which is used to pass data from the controller to the view.

The <h2>Create New Contact</h2>

HTML element displays a heading for the form.

The <form asp-action="Create">

element sets the action of the form to the Create method of the Contacts controller. This means that when the user submits the form, the data will be sent to the Create method for processing.

The various <div> and <input> elements create the input fields for the form. The <label> elements provide a description of the input field, while the <input> elements are used to actually enter data. The asp-for attribute is used to bind the input field to a property of the Contact model, so that when the form is submitted, the data will be correctly mapped to the model. The class="form-control" attribute applies Bootstrap styling to the input fields.

Finally, the <button> element is used to create a button that will submit the form data to the Create method of the Contacts controller when clicked.

**Delete.cshtml**

This is a Razor view file that displays the details of a contact and allows the user to delete the contact. Here is a step-by-step explanation of what each part of the code does:

@model Contact

This section defines the model for the view, which is the Contact model.

@{ ViewData["Title"] = "Delete"; }

This section sets the page title to "Delete".

<h2>Delete Contact</h2>

This section displays the heading "Delete Contact".

<form asp-action="DeleteConfirmed">

This section creates a form element with the asp-action attribute set to "DeleteConfirmed". This form will be submitted when the user clicks the "Delete" button.

<input type="hidden" asp-for="Id" />

This section creates a hidden input field for the contact's Id property. The value of this field will be used to identify the contact to be deleted.

<p>Are you sure you want to delete this contact type?</p>

This section displays a confirmation message asking the user if they want to delete the contact.

<d1> ... </d1>

This section creates a definition list to display the contact details. Each contact detail is displayed with a term (dt) and a description (dd).

<dt>First Name</dt><dd>@Model.FirstName</dd>

This section displays the first name of the contact.

<dt>Last Name</dt><dd>@Model.LastName</dd>

This section displays the last name of the contact.

<dt>Email</dt><dd>@Model.Email</dd>

This section displays the email of the contact.

<dt>Phone</dt><dd>@Model.Phone</dd>

This section displays the phone number of the contact.

<dt>Address</dt><dd>@Model.Address</dd>

This section displays the address of the contact.

<dt>City</dt><dd>@Model.City</dd>

This section displays the city of the contact.

<dt>State</dt><dd>@Model.State</dd>

This section displays the state of the contact.

<dt>Zip</dt><dd>@Model.Zip</dd>

This section displays the zip code of the contact.

<dt>Notes</dt><dd>@Model.Notes</dd>

This section displays any notes associated with the contact.

<dt>Contact Type</dt><dd>@Model.ContactType</dd>

This section displays the contact type of the contact.

<button type="submit" class="btn btn-danger">Delete</button>

This section adds a "Delete" button to the form. When the user clicks this button, the form will be submitted to the DeleteConfirmed action of the Contacts controller for processing.

This Razor view file generates an HTML form that allows the user to delete a contact. It starts by defining the model for the view (@model Contact) and setting the page title (ViewData["Title"] = "Delete").

The form element specifies that the form data should be submitted to the DeleteConfirmed action of the Contacts controller. It includes a hidden input for the contact's ID, which is used by the DeleteConfirmed action to identify the contact to be deleted.

The paragraph element displays a confirmation message to the user, asking if they are sure they want to delete the contact.

The d1 element creates a definition list that displays the contact's details, using the Razor syntax @Model.PropertyName to access each property of the Contact model.

Finally, the form includes a button to submit the form with a class of "btn btn-danger", indicating that this action will delete data and should be treated with caution.

**Edit.cshtml**

This is a Razor view file that allows users to delete a contact. Let's break down each part of the code and what it does:

@model Contact:

This section defines the model for the view, which in this case is the Contact model. It allows the view to access the properties of the Contact model.

@{

ViewData["Title"] = "Delete";

}:

This section sets the page title to "Delete".

<h2>Delete Contact</h2>:

This section displays a heading for the delete contact page.

<form asp-action="DeleteConfirmed">:

This section creates a form element that specifies that the form data should be submitted to the

DeleteConfirmed action of the Contacts controller.

<input type="hidden" asp-for="Id" />:

This section creates a hidden input field that is used to store the ID of the contact being deleted.

<p>Are you sure you want to delete this contact type?</p>:

This section displays a message asking the user to confirm the deletion.

<d1>:

This section creates a definition list that displays the contact details.

<dt>...</dt>:

These sections define the terms (i.e., labels) for each piece of contact information.

<dd>...</dd>:

These sections display the corresponding contact information for each label. The @Model.FirstName, @Model.LastName, etc. expressions are used to display the values of the corresponding Contact model properties.

<button type="submit" class="btn btn-danger">Delete</button>:

This section adds a button to submit the form and delete the contact. The class="btn btn-danger" attribute is used to apply Bootstrap styling to the button.

Overall, this Razor view file generates an HTML form that allows users to delete a contact by clicking a "Delete" button. When the button is clicked, the data is sent to the DeleteConfirmed action of the Contacts controller for processing.

**Index.cshtml**

This is a Razor view file that displays a list of contacts.

**Here is a step-by-step explanation of each part of the code:**

@using ExpressContactManager.Models:

This line specifies the namespace that the Contact model belongs to.

@model IEnumerable<Contact>:

This line specifies that the model used in this view is an IEnumerable of Contact objects.

ViewData["Title"] = "Contacts";:

This line sets the title of the page to "Contacts" using the ViewData dictionary.

<h2>Contacts</h2>:

This line displays the heading "Contacts" on the page.

<p><a asp-action="Create">Add New Contact</a></p>:

This line displays a link to create a new contact by linking to the Create action of the Contacts controller.

<table class="table">:

This line starts a table that will display the list of contacts.

<thead>... </thead>:

This section contains the table header, which consists of four columns: First Name, Last Name, Email, and Phone.

@if (Model != null && Model.Any()) {...}:

This code block checks if the Model is not null and has at least one element. If it does, it displays the display name of the first name, last name, email, and phone properties of the Contact model. Otherwise, it displays the display name of these properties of a new Contact object.

@if (Model != null) {...}:

This code block checks if the Model is not null. If it is not null, it displays a table row for each Contact object in the Model. For each row, it displays the first name, last name, email, and phone properties of the Contact object, as well as links to edit, view details, or delete the Contact object.

<td colspan="5" class="text-center">No contacts available.</td>:

If the Model is null, this line displays a message indicating that no contacts are available in a table cell that spans five columns and is centered.

<div class="text-center">... </div>:

This section displays a welcome message with a link to learn about building web apps with ASP.NET Core.

This Razor view file is used to display a list of contacts.

**Here's a step-by-step explanation of what each part of the code does:**

@using ExpressContactManager.Models

This line brings the Contact model into the view so it can be used

@model IEnumerable<Contact>

This line sets the model of the view to an IEnumerable of Contact objects. This means that the view expects a list of contacts to be passed to it.

ViewData["Title"] = "Contacts"

This line sets the title of the page to "Contacts".

<h2>Contacts</h2>

This line displays a header for the list of contacts.

<p><a asp-action="Create">Add New Contact</a></p>

This line displays a link that allows users to add a new contact.

<table class="table">

This line starts an HTML table that will be used to display the list of contacts.

<thead>

This line starts the table header section.

The th tags in the table header section display the column headings for the table.

@if (Model != null && Model.Any())

This line checks if the list of contacts is not empty before displaying the column headers. If the list is empty, it displays default headers for a new Contact object instead.

<tbody>

This line starts the table body section.

@if (Model != null)

This line checks if the list of contacts is not empty before displaying the contacts. If the list is empty, it displays a message saying there are no contacts available.

The foreach loop iterates over each contact in the list and displays the corresponding data in each table row.

The last column of each row contains links to edit, view details, and delete the contact.

<div class="text-center">

This line starts a section that displays a welcome message and a link to learn about building web apps with ASP.NET Core.

The HTML table is constructed to display the list of contacts. The table includes a header row with column headings for First Name, Last Name, Email, Phone, and an empty column for action links. The table body includes a loop that iterates over the collection of contacts passed to the view by the controller. For each contact, a table row is created with the values for the First Name, Last Name, Email, and Phone columns. The last column contains action links for Edit, Details, and Delete. If there are no contacts in the collection, a message is displayed in the table body indicating that there are no contacts available.

The final section is just a static HTML content that provides a welcome message and a link to learn about building web apps with ASP.NET Core.