LOVELY
PROFESSIONAL
UNIVERSITY

Report

On

ToDoList

Submitted by

Sikilammetla Sai kiran

Registration No. :- 12016732

Program Name :- B. Tech Computer Science Engineering

Under the Guidance of

Sushil Lekhi

School of Computer Science & Engineering Lovely Professional University, Phagwara

ACKNOWLEDGEMENT

I hereby declare that the work presented in this class project report entitled, “ToDoList” in subject Modern Web Programming Tools And Technique in “Computer Science & Engineering”. My extreme gratitude to Mr. Sushil Lekhi who guided us throughout the project. Without his willing disposition, spirit of accommodation,frankness, timely clarification and above all faith in us, this project could not have been completed in due time.

Introduction

In this document you are going to find what features the To-Do List System offers as well as the development process of it. They consist of the aims and the extent of development, the conceptual model, a listing of the chief concerns, and methods through which the development will be evaluated and updated.

The To-Do List System has been developed to improve people's scheduling and time management functionality by letting them generate and manage the lists of tasks effectively. The system enables people or several people to schedule their daily activities, prioritize the tasks and track the accomplishment of several tasks. As compared to ordinary methods of task management, this system would eliminate the need for lists that can be tiresome and have numerous chances of being forgotten or full of errors.

One of the biggest benefits of this system is versatility for both, personal and business utilization to enhance efficiency and save time. However, it may take some time to begin using it due to the procedural settings needed for incorporating it into team-based platforms.

From the interface point of view, the To-Do List System enables users to execute and maintain the list by simplifying the means of adding as well as organizing the coming tasks. It eliminates the need for tracking tasks with fingers and offers a systemized means of completing tasks and keeping users on schedule.

Background

To-Do List System is functionality aimed to enhance the work with tasks and increase efficiency for varying users, from unique client to entire teams. In the past, tracking of the tasks is most frequently done by using some simple paper-based lists or a simple notes capability of the operating system. But these methods are not efficient because it can lead to; late submission, no ranking system, and immense difficulty in following progress made over time.

The use of the To-Do List System also enables users to create, sort and set priorities of tasks within a virtual list depending on the available work load and the due date of the tasks. It tracks the tasks assigned, offer alerts, and helps users to set any particular task and mark its completion that improves efficiency. On account of its digitized setup the application is bidirectional which will enable those who may have tight schedules or different devices to manage the system.

Through the implementation of digital task management solutions, many organizations, and people across the globe experience improved performance. This has the following benefits, among others: reduction of the use of paper and automation of task tracking to a central point. However, the system can have setting up challenge which will take some time for the users to manage their tasks according to its system; it lacks some degree of individualistic approach in managing tasks.

In conclusion, the To-Do List System provides a clear and easy-to-follow framework for task organizing suggesting to fill the gaps left from using conventional techniques.

Problem Definition

Users who want to organize their work to become more efficient must follow proper ways of task management. The To-Do List System aims at giving a unique and effective platform to the user in creating as well as managing his/her tasks efficiently. This project is proposed to provide an easy, now, convenient, solution for organizing activities, due dates and priorities. As mentioned in its features, the To-Do List System includes setting priorities, deadlines and sorting the tasks that are easy to follow further progress. The system also has a login module to allow an individual's access to the system, where he or she requires entrance credentials. Meanwhile, the customer is either able to sign in to their account to continue with their tasks or even provide their details including name, email and password in other to get registered. The system also includes an administrative module which enable the administrator control users and configure system options. User managers can view report of the users' activity, define the global categories of tasks, and specify default reminder/notification. With the integration of an automated tracking and organizing system for tasks, people won't have to worry about forgetting which tasks they need to accomplish or if they have met a certain due date or not. Altogether, the To-Do List System solves the problems of the traditional approach to task lists and helps consumers to achieve their goals efficiently.

Objective

General objective

The pobjective of the To-Do List System project is to transition from traditional, manual task management methods to a streamlined, digital system, enabling users to manage tasks with greater efficiency and organization.

Specific Objectives

- Provide a platform for users to create, organize, and prioritize tasks in a structured format.
- Allow users to set deadlines, reminders, and categories for individual tasks.
- Automate task tracking, reducing time spent on manual updates and status checks.
- Enable users to view tasks in list or calendar formats, enhancing usability and accessibility.
- Offer secure login functionality for personalized access to task lists and settings.
- Allow administrators to monitor system usage and generate reports to analyze productivity and task completion rates.
- Provide features for prioritizing tasks, categorizing them by project, and setting recurrence for repeated tasks.
- Create a flexible system that can be used by individuals or teams to enhance productivity and achieve goals.
- Generate reports that provide insights on task completion trends, overdue tasks, and user productivity, supporting continuous improvement.

Scope of the project

The scope of this project extends beyond traditional, manual task management methods.

Key features include:

- Accessible from any location and device, as it is a digital application (user location is not restricted).
- Enables users to manage tasks without requiring constant supervision, allowing for flexible self-management.
- Designed to facilitate both end users and administrators.
- Suitable for personal, educational, or professional environments where efficient task tracking and organization are required.
- Adaptable for individual users or teams, making it versatile for various use cases.
- Provides features such as task categorization, prioritization, and deadline tracking, catering to diverse productivity needs.

This project aims to create a reliable and user-friendly system for anyone needing a more organized approach to task management.

Proposed system

Functional requirements

The To-Do List System is designed to facilitate effective task management by meeting the following functional requirements:

User Requirements

- **Administrator Aspect**
  1. Log into the system.
  2. Manage user accounts, including accepting new registrations.
  3. Add, edit, or delete tasks as required.
  4. Define and assign task categories and priorities.
  5. Generate and send task reports or productivity summaries.
  6. Set deadlines and reminders for tasks.
  7. Manage access permissions for different users.

- **User Aspect**
  1. Register and log into the system.
  2. Add and manage tasks, including setting deadlines and priorities.
  3. Organize tasks by category or project.
  4. Mark tasks as complete or review task history.
  5. Receive reminders for upcoming tasks.

- **System Analysis**
  1. Authenticate users based on username and password.
  2. Record all user activity related to task creation, updates, and completion.
  3. Generate a task history log for performance and progress tracking.
  4. Allow administrators to monitor task activity across all users.
  5. Send notifications and reminders for upcoming tasks.

Hardware Interfaces

- **Server-Side Hardware**
  1. Minimum RAM: 512 MB or more
  2. Hard Drive: 10 GB or more
  3. Communication hardware to handle client requests

- **Client-Side Hardware**
  1. Minimum RAM: 256 MB or more
  2. Communication hardware to connect with the server

Software Interface

- **Server-Side Software**
  1. Development Framework: .NET Framework
  2. Database: SQL Server
  3. Compatible Operating System: Windows

- **Client-Side Software**
  1. Web browser supporting JavaScript

Non-Functional Requirements

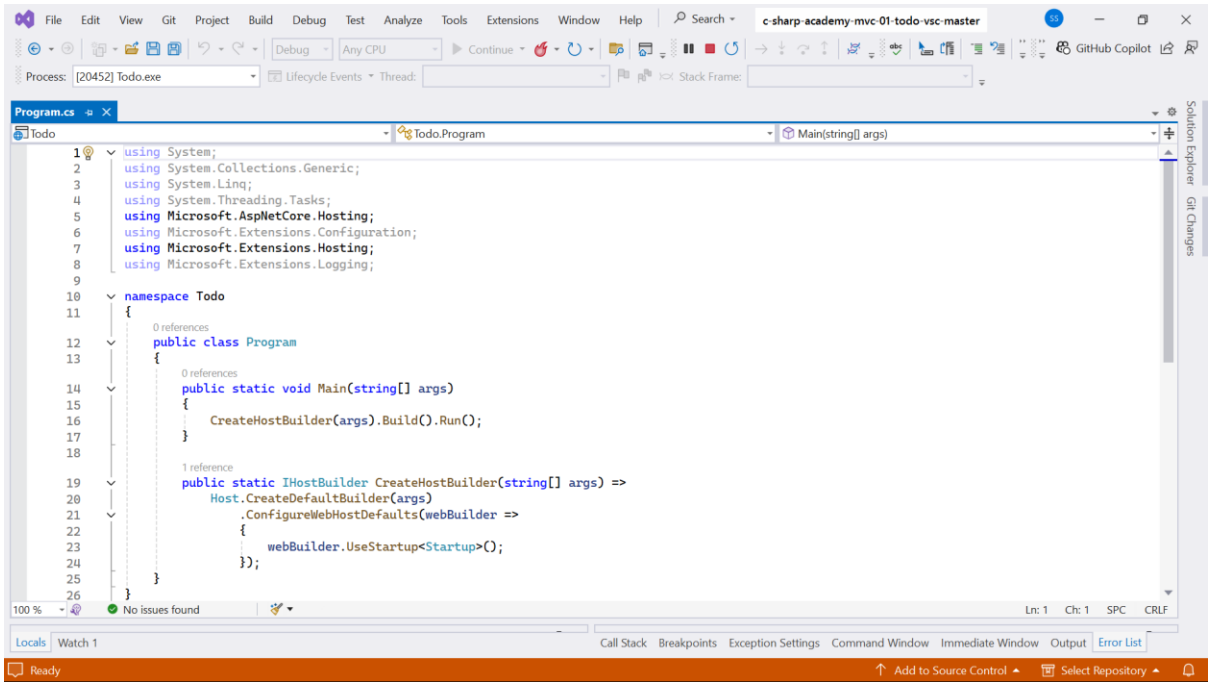
- Performance:
  1. The system should support multiple users simultaneously without degradation in performance.
  2. Task updates should be saved in real time for accuracy and reliability.

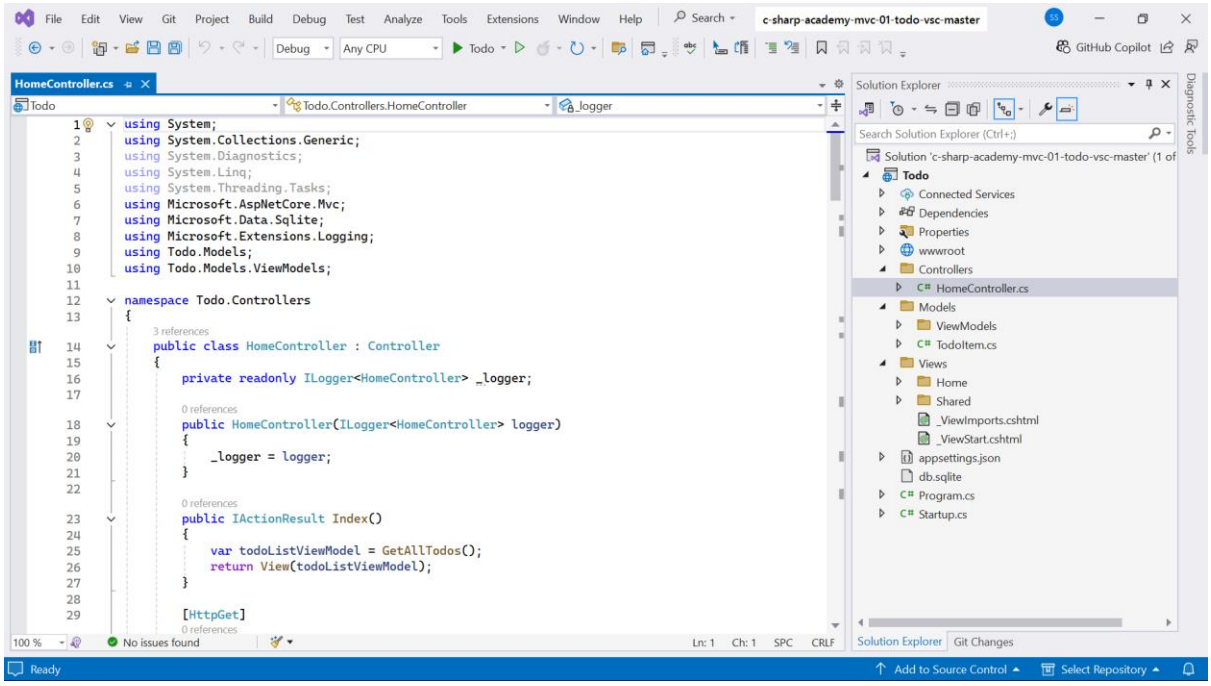
- **Usability:**
  1. The system should be intuitive and easy to navigate, minimizing the learning curve for new users.

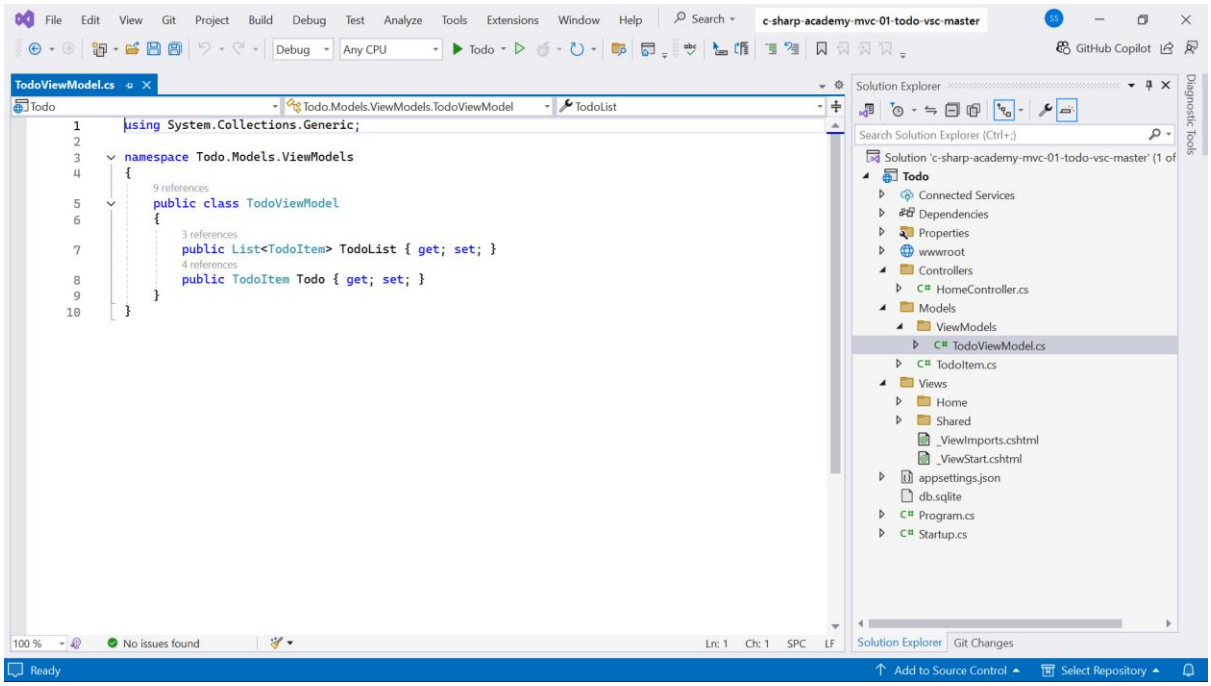
- **Portability:**
  1. Developed using HTML, CSS, C#, and JavaScript for wide compatibility.

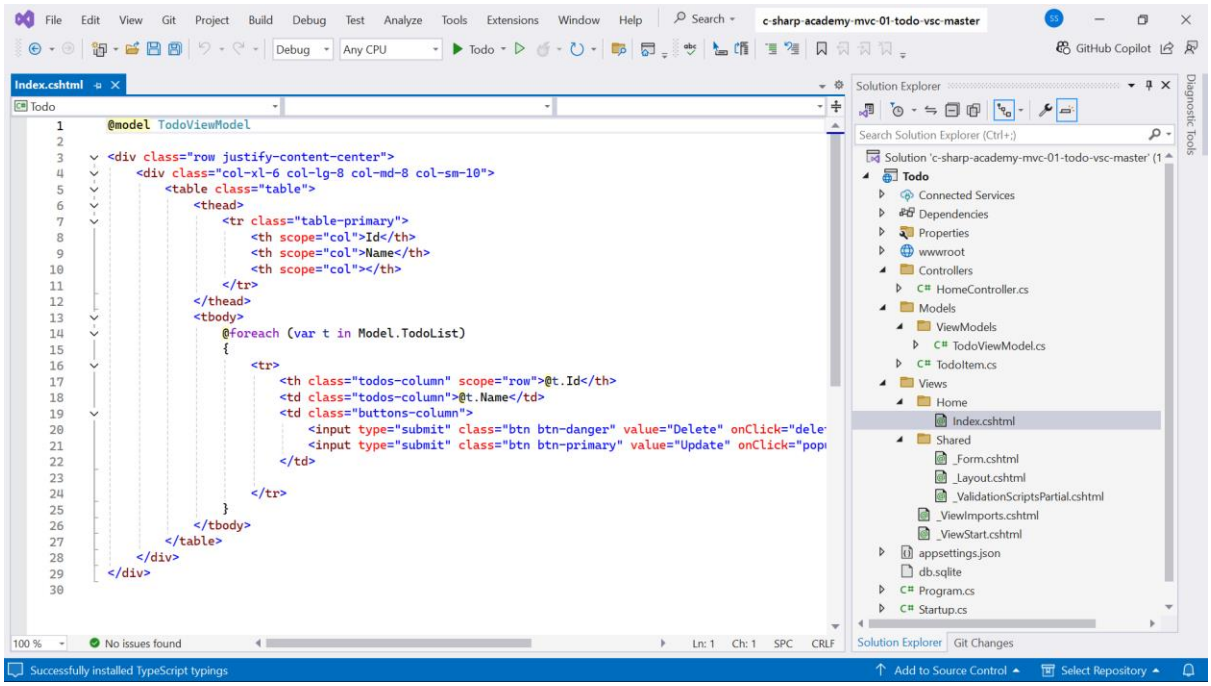
- **Availability:**
  1. The system should be accessible at any time, allowing users to add, update, or review tasks as needed.
  2. Should run on multiple operating systems, including Windows.

Screenshot’s:





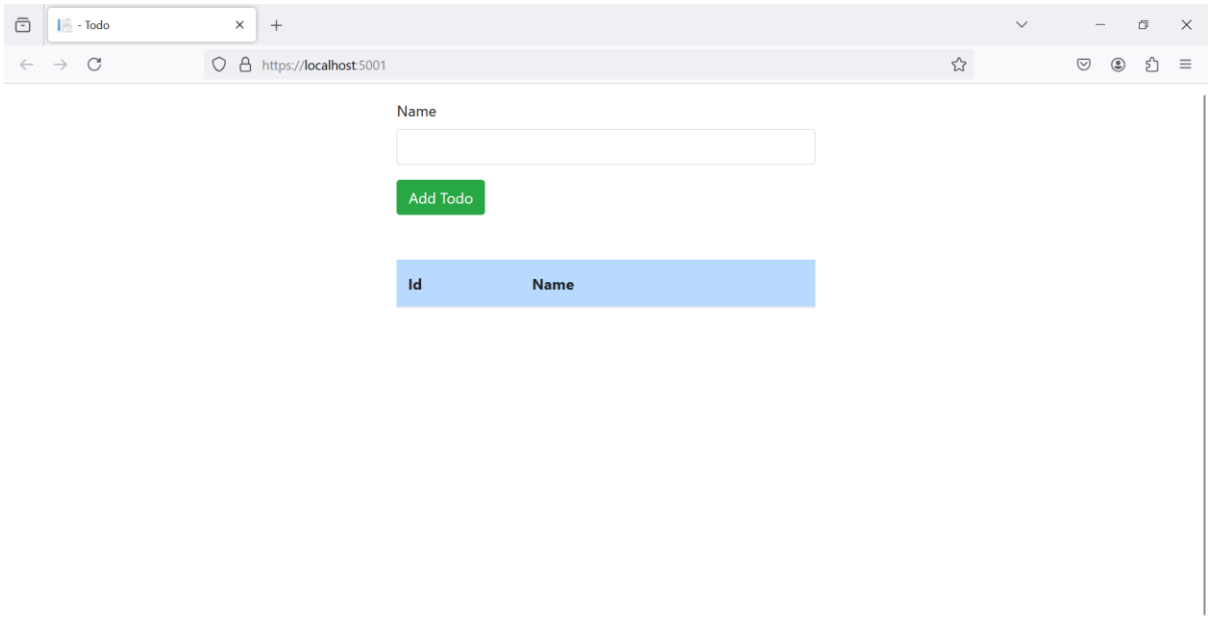

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
c-sharp-academy-mvc-01-todo-vsc-master
Debug
Any CPU
Todo
GitHub Copilot
TodoViewModel.cs
Todo
Todo.Models.ViewModels.TodoViewModel
TodoList
using System.Collections.Generic;
namespace Todo.Models.ViewModels
{
9 references
public class TodoViewModel
{
3 references
public List<TodoItem> TodoList { get; set; }
4 references
public TodoItem Todo { get; set; }
}
}
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'c-sharp-academy-mvc-01-todo-vsc-master' (1 of
Todo
Connected Services
Dependencies
Properties
wwwroot
Controllers
HomeController.cs
Models
ViewModels
TodoViewModel.cs
TodoItem.cs
Views
Home
Shared
_ViewImports.cshtml
_ViewStart.cshtml
appsettings.json
db.sqlite
Program.cs
Startup.cs
Diagnostic Tools
100 %
No issues found
Ln: 1
Ch: 1
SPC
LF
Solution Explorer
Git Changes
Ready
Add to Source Control
Select Repository

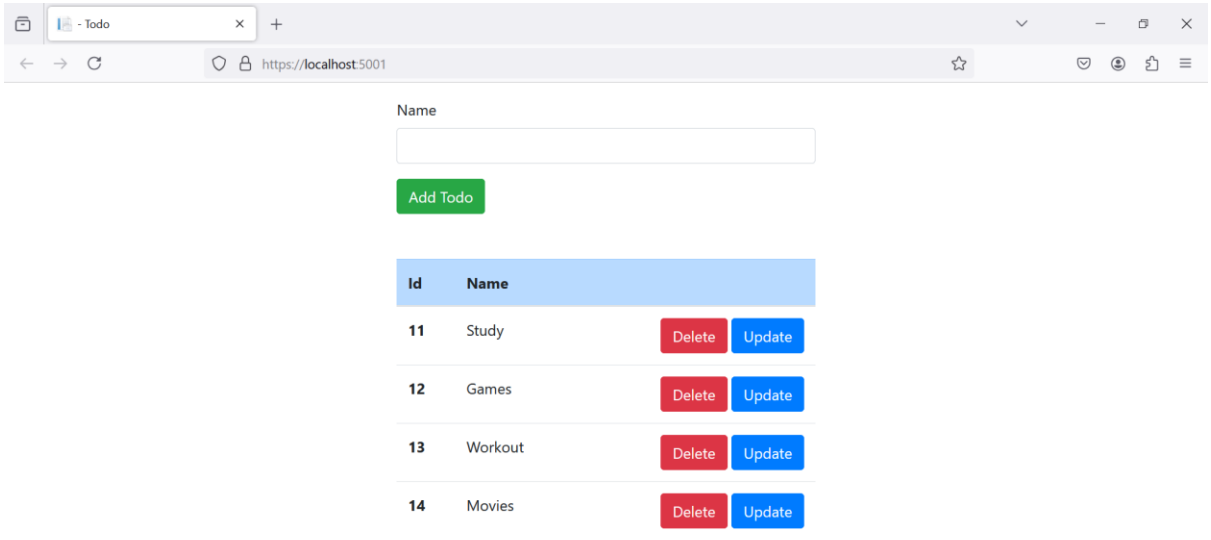

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
c-sharp-academy-mvc-01-todo-vsc-master
Debug
Any CPU
Todo
GitHub Copilot
Index.cshtml
Todo
@model TodoViewModel

<table class="table">
<thead>
<tr class="table-primary">
<th scope="col">Id</th>
<th scope="col">Name</th>
<th scope="col"></th>
</tr>
</thead>
<tbody>
@foreach (var t in Model.TodoList)
{
<tr>
<th class="todos-column" scope="row">@t.Id</th>
<td class="todos-column">@t.Name</td>
<td class="buttons-column">
<input type="submit" class="btn btn-danger" value="Delete" onClick="dele
<input type="submit" class="btn btn-primary" value="Update" onClick="popu
</td>
</tr>
}
</tbody>
</table>

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'c-sharp-academy-mvc-01-todo-vsc-master' (1
Todo
Connected Services
Dependencies
Properties
wwwroot
Controllers
HomeController.cs
Models
ViewModels
TodoViewModel.cs
TodoItem.cs
Views
Home
Index.cshtml
Shared
_Form.cshtml
_Layout.cshtml
_ValidationScriptsPartial.cshtml
_ViewImports.cshtml
_ViewStart.cshtml
appsettings.json
db.sqlite
Program.cs
Startup.cs
Diagnostic Tools
100 %
No issues found
Ln: 1
Ch: 1
SPC
CRLF
Solution Explorer
Git Changes
Successfully installed TypeScript typings
Add to Source Control
Select Repository

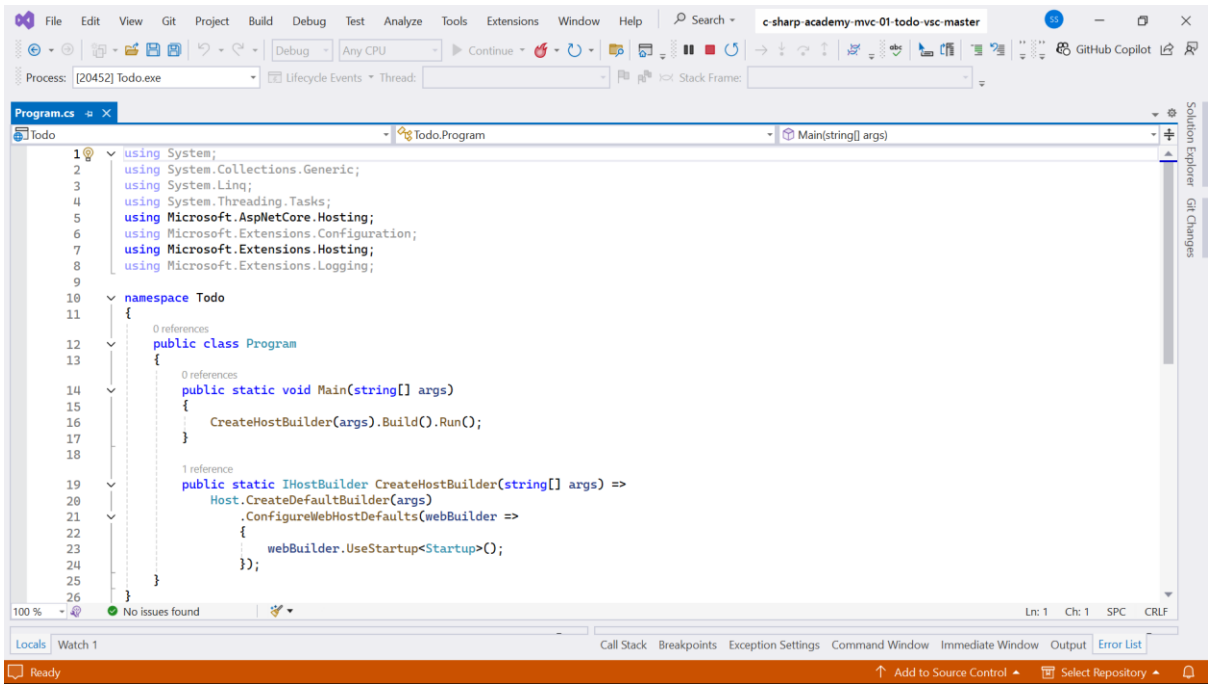
Screenshot’s of website:





Code:

1. Program.cs:




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace Todo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
```

```
            {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

Explanation:

This C# code represents the entry point for a .NET Core web application, specifically for a To-Do List system, which is designed to manage tasks within a web-based interface. This file, typically named `Program.cs`, serves as the main execution point and is responsible for configuring and running the application. Below is a detailed breakdown of each section of this code to explain its purpose and functionality in the context of a To-Do List system project.

1. Namespaces and Using Statements

```
csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
```

Purpose: The `using` statements at the top of the code file bring in various libraries and namespaces, making classes, methods, and other members available for use in the program without requiring fully qualified names.

Explanation:

`System`: Includes basic classes and base classes that define commonly used types and methods, such as `Console`, `String`, etc.

`Microsoft.AspNetCore.Hosting`: Contains classes and methods for hosting the application in a web environment, crucial for configuring and starting a web server.

`Microsoft.Extensions.Hosting`: Provides support for generic host building, allowing a more modular way of configuring and running background services, such as our web server.

`Microsoft.Extensions.Logging`: Allows logging configuration for the application, which helps in tracking events and errors during execution.

2. Namespace Definition

```
csharp
namespace Todo
{
```

Purpose: Defines a namespace called `Todo`. A namespace is a container that organizes classes and other types, helping to avoid naming conflicts.
Explanation: In this project, the `Todo` namespace likely groups together all the classes related to the To-Do List application, creating a logical structure for the codebase.

3. Program Class Definition

```
csharp
public class Program
{
```

Purpose: Declares a class named `Program`, which serves as the main entry point for this application.
Explanation: In .NET Core applications, the `Program` class conventionally contains the `Main` method, where the application starts executing. This class configures the application and hosts the web server.

4. Main Method

```
csharp
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
```

Purpose: The `Main` method is the primary entry point for the application. It is the first method called when the program is executed.
Explanation:
`Main` is a `static` method, meaning it can be called without creating an instance of the `Program` class. This is typical for entry points in C# applications.
`CreateHostBuilder(args).Build().Run()`:

`CreateHostBuilder(args)`: This method is called to set up and configure a host for the application. A host is responsible for managing the application's lifetime and essential services.

`.Build()`: Builds the `IHost` object, which represents the fully configured host for running the application.

`.Run()`: Starts the web application, keeping it active and listening for incoming HTTP requests until the application is terminated.

5. CreateHostBuilder Method

```
csharp
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Purpose: This method configures and returns an `IHostBuilder` instance, which provides a framework for configuring the host environment and services.

Explanation:

`public static IHostBuilder CreateHostBuilder(string[] args)`: This method is declared as `static` and returns an `IHostBuilder`, which will configure the application’s web server.

`Host.CreateDefaultBuilder(args)`:

This method initializes a new instance of the `HostBuilder` class with default configurations.

The default configurations include loading environment variables, reading app settings from `appsettings.json`, and enabling logging and configuration.

`.ConfigureWebHostDefaults`: This method sets up default configuration options for the web host.

`webBuilder => { webBuilder.UseStartup<Startup>(); }`:

This lambda expression takes a `webBuilder` parameter, configuring it to use a `Startup` class, which defines the specific configurations and services required to run the application.

`webBuilder.UseStartup<Startup>()`: Specifies that the application should use the `Startup` class to configure services, middleware, and request handling.

6. Explanation of Key Functionalities in Context of To-Do List System

Hosting the Application:

The `CreateHostBuilder` method configures the necessary environment and dependencies, setting up the web host to run the To-Do List application in a web environment.

Startup Class Integration:

The `UseStartup<Startup>()` method call in the `ConfigureWebHostDefaults` block links the `Startup` class to this program. The `Startup` class is where additional configurations like dependency injections, middleware, and endpoint routes are specified, which would be essential for handling requests related to the To-Do List application, such as creating, updating, and deleting tasks.

Dependency Injection and Configuration:

By using `CreateDefaultBuilder`, this setup provides a flexible and extensible architecture. The To-Do List application can easily integrate services like logging, authentication, database contexts, and configuration settings via dependency injection. For example, task management services and user authentication services could be registered within the `Startup` class.

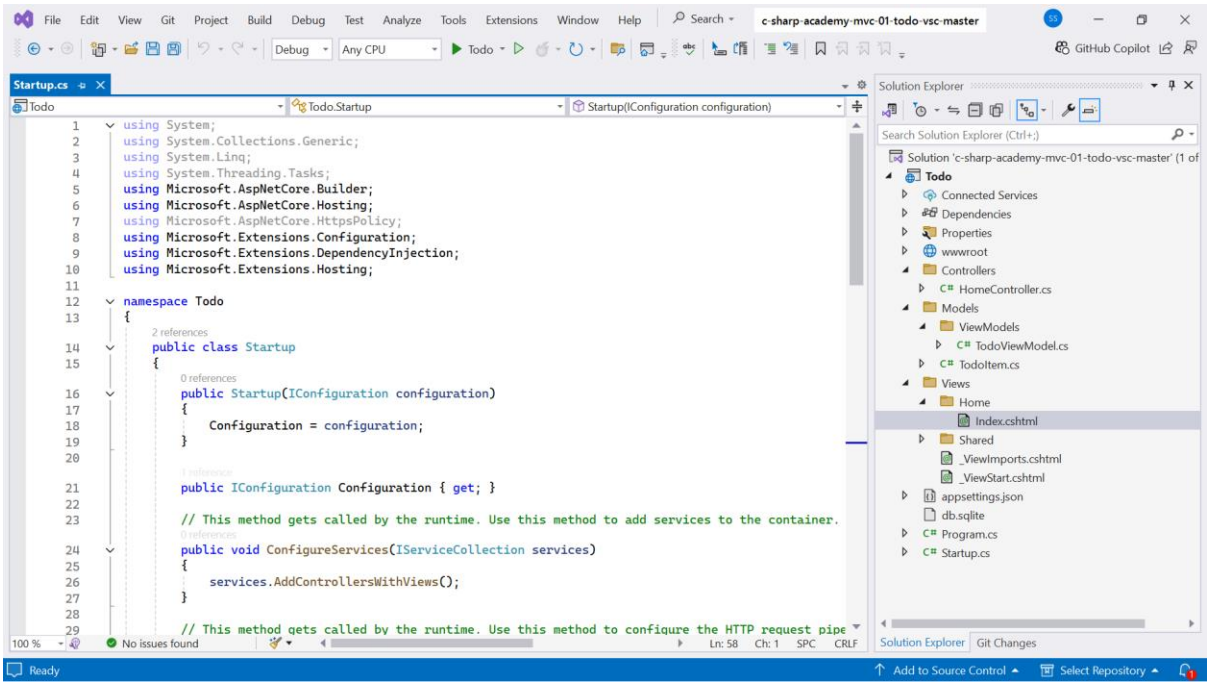
7. Significance of Each Component

Main Method with CreateHostBuilder: The main method initiates the web server and applies all configured dependencies, ensuring that the To-Do List system is accessible online.

Logging and Configuration Integration: Through the inclusion of `Microsoft.Extensions.Logging`, the program can log user activities and system events, providing insight into the To-Do List application's usage and potential issues. This feature is crucial for maintaining application reliability.

Modularity and Scalability: This structure provides a clean separation between host configuration and application-specific logic (handled by the `Startup` class), allowing the To-Do List system to be extended with additional features like notifications, user management, and data persistence.

2. Startup.cs:




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Todo
{
   public class Startup
   {
      public Startup(IConfiguration configuration)
      {
         Configuration = configuration;
      }

      public IConfiguration Configuration { get; }
```

```
        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

Explanation:

This C# code represents the `Startup` class for a .NET Core web application, specifically designed to manage the configuration and setup of services and the HTTP request pipeline. The `Startup` class is essential for defining how the application should behave in different environments (e.g., Development or Production) and configuring services that the application will use. Below is an in-depth explanation of each section of the code in the context of a To-Do List system project.

1. Using Statements

```
csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

Purpose: The `using` directives bring in essential libraries and namespaces, making classes, methods, and types available to the code.

Explanation:

`Microsoft.AspNetCore.Builder`: Contains classes and methods for building HTTP request pipelines.

`Microsoft.AspNetCore.Hosting`: Used for configuring and managing the application's hosting environment.

`Microsoft.AspNetCore.HttpsPolicy`: Supports HTTP policies, such as enforcing HTTPS.

`Microsoft.Extensions.Configuration`: Allows access to configuration settings, such as those defined in `appsettings.json`.

`Microsoft.Extensions.DependencyInjection`: Enables dependency injection, which is the way services are registered and made available throughout the application.

`Microsoft.Extensions.Hosting`: Provides support for the application's hosting, allowing configuration for different environments.

2. Namespace Definition
```
csharp
namespace Todo
{
```

Purpose: This code defines a namespace `Todo` that contains all classes related to the To-Do List application.
Explanation: Grouping the classes related to the To-Do List project in a single namespace helps organize the code and avoid naming conflicts, especially in larger projects.

3. Startup Class Definition
```
csharp
public class Startup
{
```

Purpose: Declares the `Startup` class, which is central to configuring the application's services and request handling pipeline.
Explanation: The `Startup` class is automatically loaded by .NET Core when the application starts. It provides two main methods, `ConfigureServices` and `Configure`, that are essential for setting up the application.

4. Constructor
```
csharp
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

Purpose: Initializes the `Startup` class with a configuration object.
Explanation:
`Startup(IConfiguration configuration)`: This constructor injects an `IConfiguration` object, which is used to access application settings from sources like `appsettings.json` or environment variables.

`public IConfiguration Configuration { get; }`: Stores the configuration instance as a property, making it accessible within the class. This allows access to settings such as connection strings or feature flags, which may be used to customize aspects of the To-Do List application.

5. ConfigureServices Method

```
csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
}
```

Purpose: This method is called by the runtime to register services that the application will use.

Explanation:

`IServiceCollection services`: The `services` parameter allows for dependency injection, making it possible to add various services to the application's container.

`services.AddControllersWithViews()`: Registers support for MVC (Model-View-Controller) with controllers and views. This is essential for the To-Do List system, as it allows the application to handle HTTP requests and return web pages or JSON data. Controllers can be used to handle tasks such as creating, updating, or deleting to-do items.

In Context of the To-Do List Project: Adding controllers and views provides the fundamental setup for routing and response handling, so users can interact with the application, manage tasks, and view pages effectively.

6. Configure Method

```
csharp
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
```

Purpose: Configures the HTTP request pipeline, which determines how requests are handled as they move through the application.

Explanation: The `Configure` method uses middleware components to handle incoming HTTP requests and manage responses. Each middleware component processes the request and can pass it along the pipeline.

7. Environment-Based Configuration

```
csharp
```

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}
```

Purpose: Sets up different behaviors based on whether the application is running in a development or production environment.

Explanation:

`env.IsDevelopment()`: Checks if the application is in the Development environment. If true, it enables the Developer Exception Page, which provides detailed error information for easier debugging.

`app.UseExceptionHandler("/Home/Error")`: In a production environment, directs errors to a specific error-handling page (`/Home/Error`).

`app.UseHsts()`: Enforces HTTP Strict Transport Security (HSTS) for security by forcing HTTPS for future requests.

In Context of the To-Do List Project: This ensures that in a live environment, the To-Do List application handles errors gracefully and secures user data by enforcing HTTPS.

8. Other Middleware Components

```
csharp
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();
```

Purpose: Sets up middleware to handle specific aspects of the request pipeline.

Explanation:

`app.UseHttpsRedirection()`: Redirects HTTP requests to HTTPS, enhancing security.

`app.UseStaticFiles()`: Enables serving of static files like CSS, JavaScript, and images, which are crucial for the front end of the To-Do List application.

`app.UseRouting()`: Activates routing, enabling the application to determine which controller and action to invoke based on the URL.

`app.UseAuthorization()`: Enables authorization checks, ensuring that certain areas or actions are restricted to authorized users.

In Context of the To-Do List Project: HTTPS redirection and authorization are critical for securing user data, while static files and routing are necessary to ensure the application is both functional and visually responsive.

9. Endpoint Configuration

```
csharp
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Purpose: Defines the endpoint routing configuration for the application.

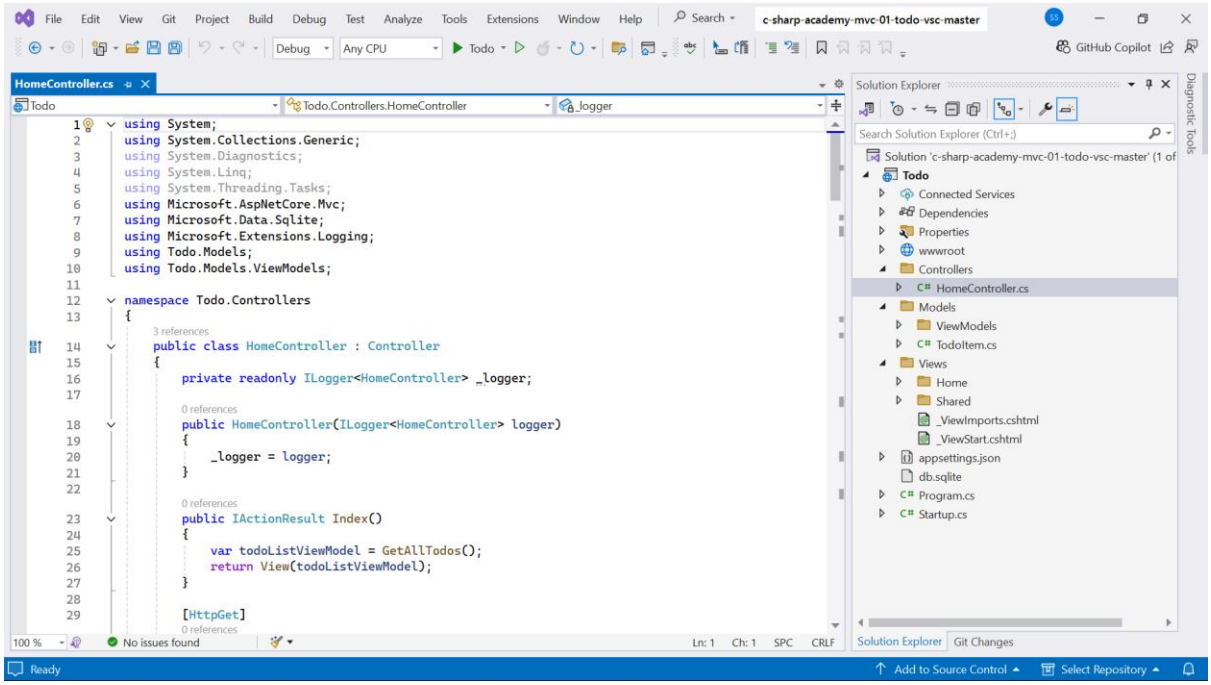
Explanation:

`MapControllerRoute`: Sets up a default route for the MVC controllers. In this case, if no specific route is provided, the application will default to the `Home` controller and the `Index` action.

`{controller=Home}/{action=Index}/{id?}`: This pattern defines the route format, where the `controller` is `Home`, `action` is `Index`, and `id` is optional.

In Context of the To-Do List Project: The routing setup allows users to access different areas of the application, such as creating new to-do items or viewing a list of tasks, by using meaningful URLs. For example, `/Todo/List` might show all tasks, while `/Todo/Add` might open a form to create a new task.

3. HomeController.cs:




```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Data.Sqlite;
using Microsoft.Extensions.Logging;
using Todo.Models;
using Todo.Models.ViewModels;

namespace Todo.Controllers
{
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;

        public HomeController(ILogger<HomeController> logger)
        {
            _logger = logger;
        }

        public IActionResult Index()
```

```
{
    var todoListViewModel = GetAllTodos();
    return View(todoListViewModel);
}

[HttpGet]
public JsonResult PopulateForm(int id)
{
    var todo = GetById(id);
    return Json(todo);
}


internal TodoViewModel GetAllTodos()
{
    List<TodoItem> todoList = new();

    using (SqliteConnection con =
          new SqliteConnection("Data Source=db.sqlite"))
    {
        using (var tableCmd = con.CreateCommand())
        {
            con.Open();
            tableCmd.CommandText = "SELECT * FROM todo";

            using (var reader = tableCmd.ExecuteReader())
            {
                if (reader.HasRows)
                {
                    while (reader.Read())
                    {
                        todoList.Add(
                            new TodoItem
                            {
                                Id = reader.GetInt32(0),
                                Name = reader.GetString(1)
                            });
                    }
                }
                else
```

```
                {
                    return new TodoViewModel
                    {
                        TodoList = todoList
                    };
                }
            };
        }
    }

    return new TodoViewModel
    {
        TodoList = todoList
    };
}

internal TodoItem GetById(int id)
{
    TodoItem todo = new();

    using (var connection =
        new SqliteConnection("Data Source=db.sqlite"))
    {
        using (var tableCmd = connection.CreateCommand())
        {
            connection.Open();
            tableCmd.CommandText = $"SELECT * FROM todo Where Id =
'{id}'";

            using (var reader = tableCmd.ExecuteReader())
            {
                if (reader.HasRows)
                {
                    reader.Read();
                    todo.Id = reader.GetInt32(0);
                    todo.Name = reader.GetString(1);
                }
                else
                {
                    return todo;
```

```
                }
            };
        }
    }

    return todo;
}

public RedirectResult Insert(TodoItem todo)
{
    using (SqliteConnection con =
         new SqliteConnection("Data Source=db.sqlite"))
    {
        using (var tableCmd = con.CreateCommand())
        {
            con.Open();
            tableCmd.CommandText = $"INSERT INTO todo (name)
VALUES ('{todo.Name}')";
            try
            {
                tableCmd.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
    return Redirect("https://localhost:5001/");
}

[HttpPost]
public JsonResult Delete(int id)
{
    using (SqliteConnection con =
         new SqliteConnection("Data Source=db.sqlite"))
    {
        using (var tableCmd = con.CreateCommand())
        {
            con.Open();
```

```
                tableCmd.CommandText = $"DELETE from todo WHERE Id =
'{id}'";
                tableCmd.ExecuteNonQuery();
            }
        }

        return Json(new {});
    }

    public RedirectResult Update(TodoItem todo)
    {
        using (SqliteConnection con =
            new SqliteConnection("Data Source=db.sqlite"))
        {
            using (var tableCmd = con.CreateCommand())
            {
                con.Open();
                tableCmd.CommandText = $"UPDATE todo SET name =
'{todo.Name}' WHERE Id = '{todo.Id}'";
                try
                {
                    tableCmd.ExecuteNonQuery();
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }

        return Redirect("https://localhost:5001/");
    }
  }
}
```

Explanation:

This C# code represents the `HomeController` class, which is responsible for handling HTTP requests, managing interactions with a SQLite database, and

performing CRUD operations for a To-Do List application. The controller uses dependency injection, integrates logging, and provides methods for viewing, adding, updating, and deleting to-do items. Below is an in-depth explanation of each section of the code in the context of a To-Do List system project.

1. Using Statements

```
csharp
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Data.Sqlite;
using Microsoft.Extensions.Logging;
using Todo.Models;
using Todo.Models.ViewModels;
```

Purpose: Imports essential namespaces and libraries for handling HTTP requests, logging, and database interaction.

Explanation:

`Microsoft.AspNetCore.Mvc`: Enables MVC-related functionality such as controllers and views.

`Microsoft.Data.Sqlite`: Provides functionality for SQLite database operations.

`Microsoft.Extensions.Logging`: Allows logging events for debugging and tracking purposes.

`Todo.Models` and `Todo.Models.ViewModels`: Allows access to the application's custom models, like `TodoItem` and `TodoViewModel`, which are essential for managing and displaying to-do items.

2. Namespace Definition

```
csharp
namespace Todo.Controllers
{
```

Purpose: Defines a namespace for the controllers in the To-Do List application.

Explanation: Organizing controllers within a specific namespace (`Todo.Controllers`) helps structure the code, especially as the project grows.

3. HomeController Class Definition

csharp

```
public class HomeController : Controller
{
```

Purpose: Defines the `HomeController` class that inherits from the `Controller` base class in ASP.NET Core.

Explanation: As the main controller, `HomeController` is responsible for handling user requests, processing data, and rendering views in the To-Do List application.

4. Logger Dependency Injection

csharp

```
private readonly ILogger<HomeController> _logger;

public HomeController(ILogger<HomeController> logger)
{
    _logger = logger;
}
```

Purpose: Initializes a logger to enable logging throughout the controller.

Explanation:

`ILogger<HomeController> logger`: Injected via dependency injection, allowing the controller to log events, errors, and other important information.

In Context of the To-Do List Project: This logging setup is valuable for debugging issues or logging errors when interacting with the SQLite database.

5. Index Method

csharp

```
public IActionResult Index()
{
    var todoListViewModel = GetAllTodos();
    return View(todoListViewModel);
}
```

Purpose: Renders the main page, displaying the list of to-do items.

Explanation:

`GetAllTodos()`: Calls an internal method to retrieve all to-do items from the database.

`return View(todoListViewModel)`: Passes the to-do list data to the view to be displayed to the user.

In Context of the To-Do List Project: The main view displays all tasks, making it the starting point for users interacting with the to-do list.

6. PopulateForm Method (HTTP GET)

csharp

```
[HttpGet]
public JsonResult PopulateForm(int id)
{
    var todo = GetById(id);
    return Json(todo);
}
```

Purpose: Returns the details of a specific to-do item in JSON format.

Explanation:

`GetById(id)`: Retrieves a to-do item by its ID from the database.

`return Json(todo)`: Returns the retrieved item as JSON, enabling front-end interactions such as form population.

In Context of the To-Do List Project: This method allows the application to populate form fields with the details of a specific task, enabling easier editing for the user.

7. GetAllTodos Method

csharp

```
internal TodoViewModel GetAllTodos()
{
    List<TodoItem> todoList = new();
    using (SqliteConnection con = new SqliteConnection("Data
Source=db.sqlite"))
    {
        using (var tableCmd = con.CreateCommand())
        {
            con.Open();
            tableCmd.CommandText = "SELECT * FROM todo";
            using (var reader = tableCmd.ExecuteReader())
            {
                if (reader.HasRows)
                {
                    while (reader.Read())
```

```
                {
                    todoList.Add(new TodoItem { Id = reader.GetInt32(0), Name
= reader.GetString(1) });
                }
            }
        }
    }
    return new TodoViewModel { TodoList = todoList };
}
```

Purpose: Retrieves all to-do items from the database.
Explanation:
Connects to the SQLite database and executes a SQL query to fetch all rows from the `todo` table.
Returns a `TodoViewModel` containing a list of all to-do items.
In Context of the To-Do List Project: This method is essential for displaying the full list of tasks on the main page.

8. GetById Method

```
csharp
internal TodoItem GetById(int id)
{
    TodoItem todo = new();
    using (var connection = new SqliteConnection("Data Source=db.sqlite"))
    {
        using (var tableCmd = connection.CreateCommand())
        {
            connection.Open();
            tableCmd.CommandText = $"SELECT * FROM todo Where Id =
'{id}'";
            using (var reader = tableCmd.ExecuteReader())
            {
                if (reader.HasRows)
                {
                    reader.Read();
                    todo.Id = reader.GetInt32(0);
                    todo.Name = reader.GetString(1);
                }
            }
```

```
        }
    }
    return todo;
}
```

Purpose: Retrieves a specific to-do item by ID from the database.
Explanation: Uses a SQL query to fetch a single to-do item and populate its `Id` and `Name` properties.
In Context of the To-Do List Project: This method is used for operations like viewing or editing a specific task.

9. Insert Method

```
csharp
public RedirectResult Insert(TodoItem todo)
{
    using (SqliteConnection con = new SqliteConnection("Data
Source=db.sqlite"))
    {
        using (var tableCmd = con.CreateCommand())
        {
            con.Open();
            tableCmd.CommandText = $"INSERT INTO todo (name) VALUES
('{todo.Name}')";
            tableCmd.ExecuteNonQuery();
        }
    }
    return Redirect("https://localhost:5001/");
}
```

Purpose: Adds a new to-do item to the database.
Explanation: Executes an `INSERT` SQL command to add a new row to the `todo` table, then redirects to the home page.
In Context of the To-Do List Project: This method allows users to add new tasks to their list.

10. Delete Method (HTTP POST)

```
csharp
[HttpPost]
public JsonResult Delete(int id)
{
```

```
    using (SqliteConnection con = new SqliteConnection("Data
Source=db.sqlite"))
    {
      using (var tableCmd = con.CreateCommand())
      {
        con.Open();
        tableCmd.CommandText = $"DELETE from todo WHERE Id =
'{id}'";
        tableCmd.ExecuteNonQuery();
      }
    }
    return Json(new {});
  }
```

Purpose: Deletes a specific to-do item from the database.
Explanation: Executes a `DELETE` SQL command to remove a to-do item based on its ID.
In Context of the To-Do List Project: This method provides functionality for users to delete tasks.

11. Update Method

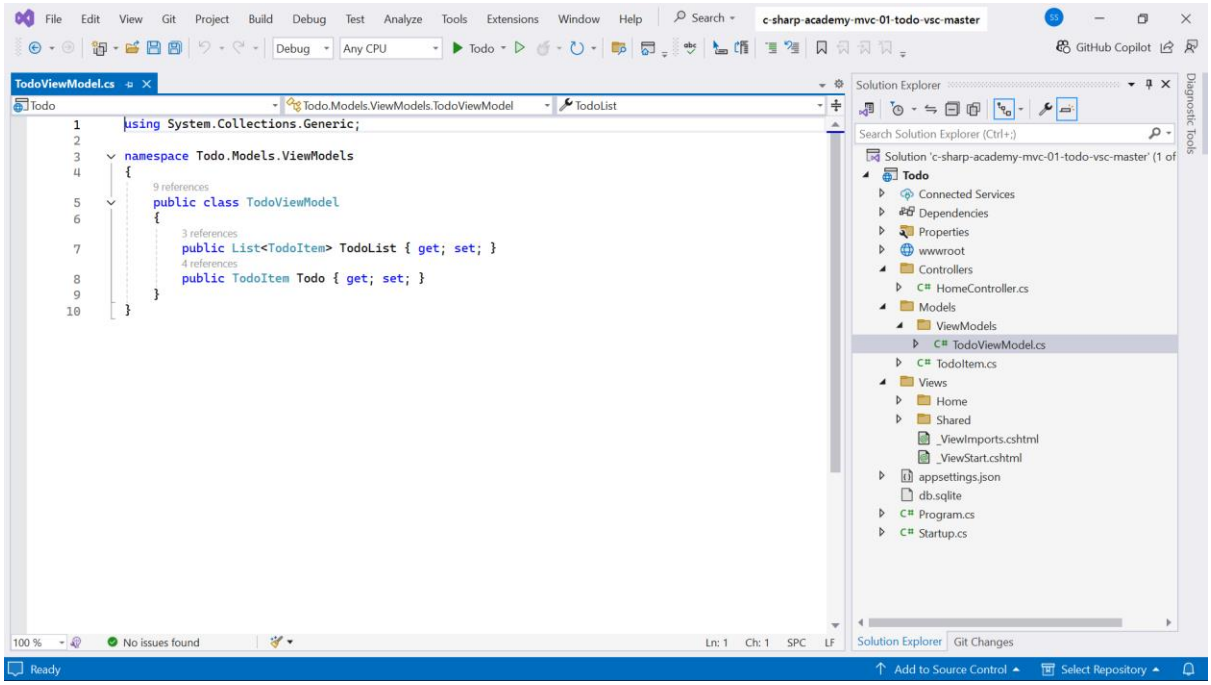
```
csharp
public RedirectResult Update(TodoItem todo)
{
    using (SqliteConnection con = new SqliteConnection("Data
Source=db.sqlite"))
    {
      using (var tableCmd = con.CreateCommand())
      {
        con.Open();
        tableCmd.CommandText = $"UPDATE todo SET name =
'{todo.Name}' WHERE Id = '{todo.Id}'";
        tableCmd.ExecuteNonQuery();
      }
    }
    return Redirect("https://localhost:5001/");
  }
```

Purpose: Updates an existing to-do item in the database.

Explanation: Executes an `UPDATE` SQL command to modify an existing item's name based on its ID.

In Context of the To-Do List Project: Allows users to edit the name of an existing task in their list.

4. TodoViewModel.cs:




```
using System.Collections.Generic;

namespace Todo.Models.ViewModels
{
    public class TodoViewModel
    {
        public List<TodoItem> TodoList { get; set; }
        public TodoItem Todo { get; set; }
    }
}
```

Explanation:

This C# code defines the `TodoViewModel` class, which is part of the `Todo.Models.ViewModels` namespace. This class is a view model, which means it serves as a container to hold data that the view (user interface) can use

to display information. The `TodoViewModel` class includes properties that make it easier to work with and display the list of to-do items in a view.

1. Using Statement
   csharp
   using System.Collections.Generic;

   Purpose: Imports the `System.Collections.Generic` namespace.
   Explanation: `System.Collections.Generic` provides access to collections like `List<T>`, `Dictionary<TKey, TValue>`, and other generic data structures, which enable more flexible data handling.
   In Context of the To-Do List Project: This namespace is necessary for the `List<TodoItem>` property within the view model, allowing the application to hold a list of `TodoItem` objects.

2. Namespace Definition
   csharp
   namespace Todo.Models.ViewModels
   {

   Purpose: Defines the namespace for the view model class.
   Explanation: `Todo.Models.ViewModels` is a logical grouping for view model classes related to the To-Do List application. This helps in organizing code and separating view models from other models and controller logic.
   In Context of the To-Do List Project: Using a distinct namespace for view models makes it clear which classes are responsible for transferring data between the controller and the view.

3. TodoViewModel Class Definition
   csharp
   public class TodoViewModel
   {

   Purpose: Defines a public `TodoViewModel` class, which serves as a view model in the application.
   Explanation: A view model encapsulates data required by the view, so it’s easier to manage the data flow between the backend and the UI.
   In Context of the To-Do List Project: `TodoViewModel` helps gather all the data needed to display the to-do items on the UI, so the view can easily bind to the properties in this model.

4. TodoList Property

csharp

public List<TodoItem> TodoList { get; set; }

Purpose: Stores a list of `TodoItem` objects representing multiple to-do items.

Explanation:

`public`: Accessible to other parts of the application, especially the view.

`List<TodoItem>`: A collection that holds multiple `TodoItem` objects.

`get; set;`: Auto-implemented property, allowing the list to be retrieved or modified.

In Context of the To-Do List Project: This property is used to display the entire list of tasks to the user. It's often accessed in the main view where all to-do items are shown.

5. Todo Property

csharp

public TodoItem Todo { get; set; }

Purpose: Stores a single `TodoItem` object representing a single to-do item.

Explanation:

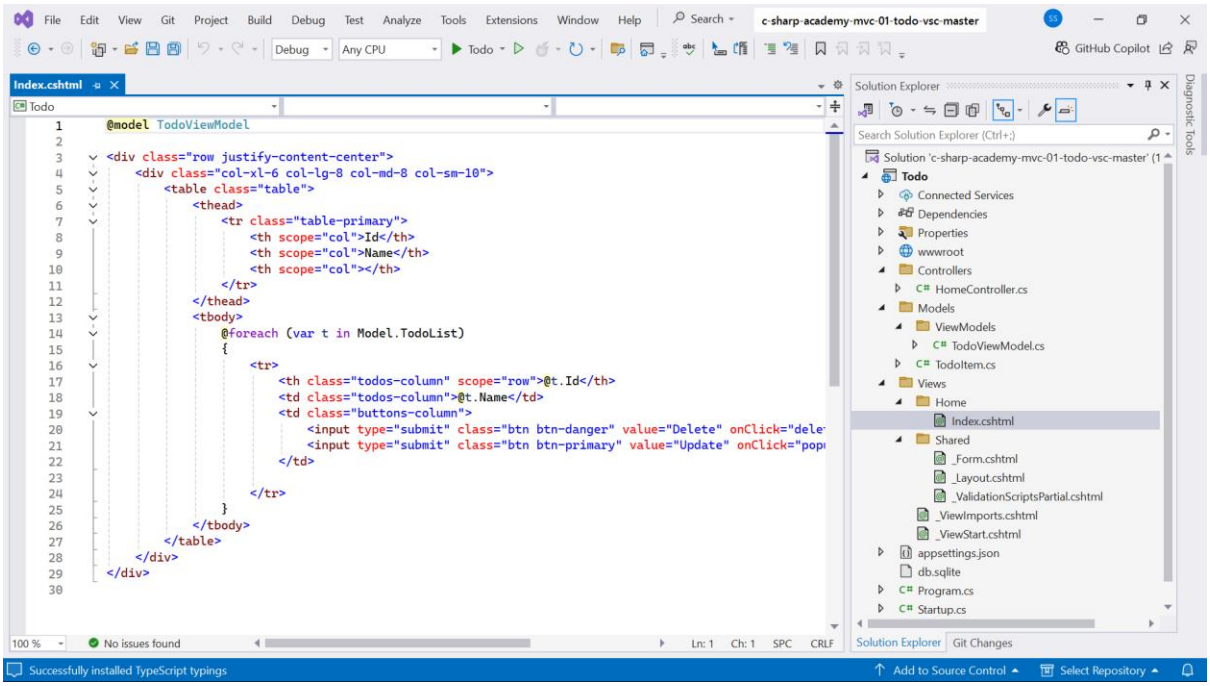
`public`: Accessible to other parts of the application, especially the view.

`TodoItem`: Represents a single to-do item.

`get; set;`: Auto-implemented property, allowing the item to be retrieved or modified.

In Context of the To-Do List Project: This property is used for displaying or editing a single task, such as when populating a form to update or delete an individual item.

5. Index.cshtml:




```
@model TodoViewModel

<div class="row justify-content-center">
   <div class="col-xl-6 col-lg-8 col-md-8 col-sm-10">
      <table class="table">
         <thead>
            <tr class="table-primary">
               <th scope="col">Id</th>
               <th scope="col">Name</th>
               <th scope="col"></th>
            </tr>
         </thead>
         <tbody>
            @foreach (var t in Model.TodoList)
            {
               <tr>
                  <th class="todos-column" scope="row">@t.Id</th>
                  <td class="todos-column">@t.Name</td>
                  <td class="buttons-column">
                     <input type="submit" class="btn btn-danger" value="Delete"
onClick="deleteTodo(@t.Id)" />
```

```
                        <input type="submit" class="btn btn-primary"
value="Update" onClick="populateForm(@t.Id)" />
                    </td>

                </tr>
            }
        </tbody>
    </table>
  </div>
</div>
```

Explanation:

This code is an ASP.NET Razor view written in C#, specifically for rendering a list of to-do items stored in a `TodoViewModel` object in an HTML table format. Each to-do item has options for updating or deleting it, making the user interface interactive. Let's go through each part of the code to understand its function and purpose.

1. @model Directive
   csharp
   @model TodoViewModel

   Purpose: Specifies the type of data that this Razor view will use.
   Explanation: The `@model` directive indicates that this view expects a `TodoViewModel` object. This model contains a list of `TodoItem` objects (`TodoList`) that will be displayed on the page.
   In Context of the To-Do List Project: This allows the view to access properties of `TodoViewModel`, particularly the `TodoList` property, to display the list of to-do items.

2. Container Divs
   html

   Purpose: Provides responsive layout styling and centers the content within the view.

Explanation: These `div` elements use Bootstrap classes (`row`, `justify-content-center`, etc.) to create a responsive layout, with the to-do list centered on the screen and optimized for different screen sizes.

In Context of the To-Do List Project: This responsive layout ensures that the table of to-do items is displayed in a user-friendly way across various devices.

3. HTML Table Structure

```
html
<table class="table">
    <thead>
        <tr class="table-primary">
            <th scope="col">Id</th>
            <th scope="col">Name</th>
            <th scope="col"></th>
        </tr>
    </thead>
```

Purpose: Sets up a table to display each to-do item's ID and Name with a header row.

Explanation:

The `<table class="table">` element creates a table with Bootstrap styling.

`<thead>` and `<tr>` define the header row, and `table-primary` class provides a background color.

Each `<th scope="col">` represents a header cell for each column: "Id", "Name," and an empty header for action buttons.

In Context of the To-Do List Project: This layout clearly separates the table into columns for displaying each to-do item's details and provides space for action buttons.

4. Looping Through the To-Do List

```
html
<tbody>
    @foreach (var t in Model.TodoList)
    {
        <tr>
            <th class="todos-column" scope="row">@t.Id</th>
            <td class="todos-column">@t.Name</td>
            <td class="buttons-column">
                <input type="submit" class="btn btn-danger" value="Delete"
onClick="deleteTodo(@t.Id)" />
```

```
                <input type="submit" class="btn btn-primary" value="Update"
onClick="populateForm(@t.Id)" />
            </td>
        </tr>
    }
</tbody>
```

Purpose: Iterates over each `TodoItem` in `Model.TodoList` and generates a row in the table for each item.

Explanation:

`@foreach (var t in Model.TodoList)` loops through each `TodoItem` in the `TodoList`.

`<tr>` represents a new row in the table for each item, displaying the item's `Id` and `Name`.

Each `<td>` displays data within the row, using the `@t.Id` and `@t.Name` Razor expressions to bind the data.

In Context of the To-Do List Project: This dynamically generates rows for each to-do item in the list, making the table content responsive to the data in `TodoViewModel`.

5. Action Buttons

html

```
<input type="submit" class="btn btn-danger" value="Delete"
onClick="deleteTodo(@t.Id)" />
<input type="submit" class="btn btn-primary" value="Update"
onClick="populateForm(@t.Id)" />
```

Purpose: Provides buttons for deleting and updating each to-do item.

Explanation:

The `Delete` button has a `btn btn-danger` Bootstrap class for a red-colored button, indicating a destructive action.

The `Update` button has a `btn btn-primary` class for a blue-colored button, indicating an action to modify data.

`onClick="deleteTodo(@t.Id)"` triggers the `deleteTodo` JavaScript function with the `Id` of the current item.

`onClick="populateForm(@t.Id)"` triggers the `populateForm` function, loading the item details into a form for editing.

In Context of the To-Do List Project: These buttons allow the user to directly interact with the to-do items, deleting or updating them as needed.

Conclusion:

To-Do List System can easily be implemented to offer the best means of creating, organizing and completing tasks. This system combines the manual ways of working with a modern digital solution that centralizes to-do lists to let the users easily create and prioritize their tasks and track them. To this end, the enhancements are as follows; secure login, categorization, deadlines and reminders – this helps a user manage his/her workload effectively without missing deadlines.

For administrators it can provide strong monitoring and reporing capabilities which extends well to both individual and group usage. Also this convenience as a system that could be accessed instead of being operated, and from any device further increases its functionality. All in all, this project positively solves the existing problem of the lack of availability and automation in task management, and demonstrates a widely applicable and dependable application for users of all types.