Kuwait University
College of Computing Sciences and Engineering
Department of Information Science

COMP 2139
Web Application Development with C# .NET

Lab 4
Contact Manager Application
Controllers and Views

Table of Contents

Laboratory #4 – Contact Manager Application 3

Laboratory #4 – Developing a Single Page Web Application

1. Laboratory Objective

The goal of this lab is to create a Contact Manager Application using C# .NET – Controller and Views.

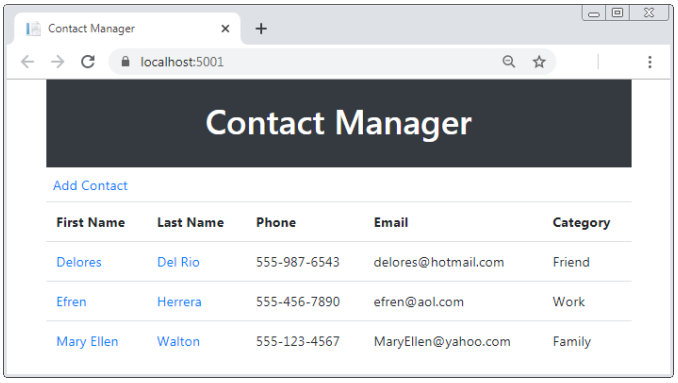
2. Laboratory Learning Outcomes: After conducting this laboratory students will be able to:

a. Create and maintain Git repository project
b. Create a data-drive web application using C#.NET
c. Constructing controllers and Views

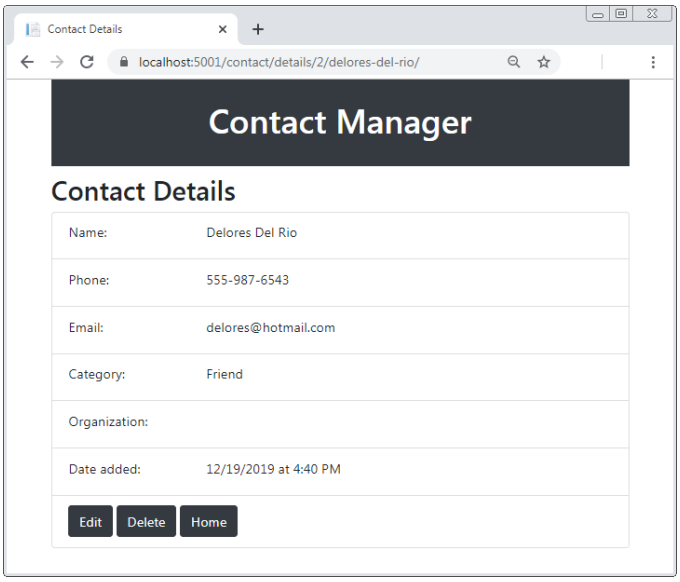
3. Laboratory Instructions

For this lab, we will continue with the implementation of a multi-page, data driven application modeling the wireframes below:

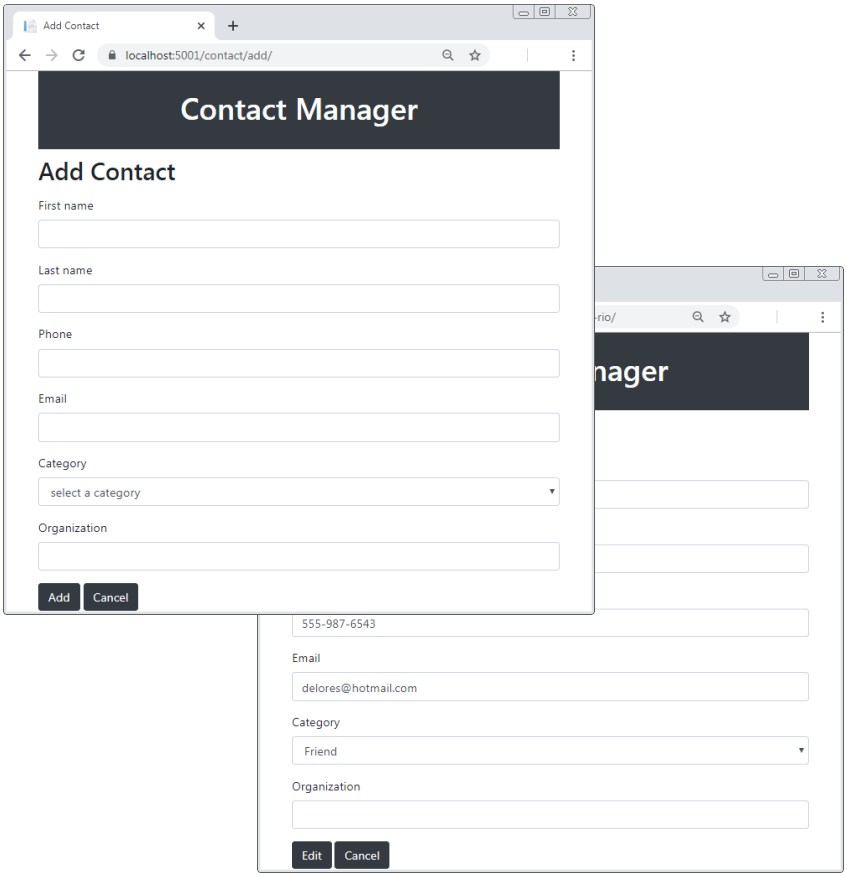
The Home page



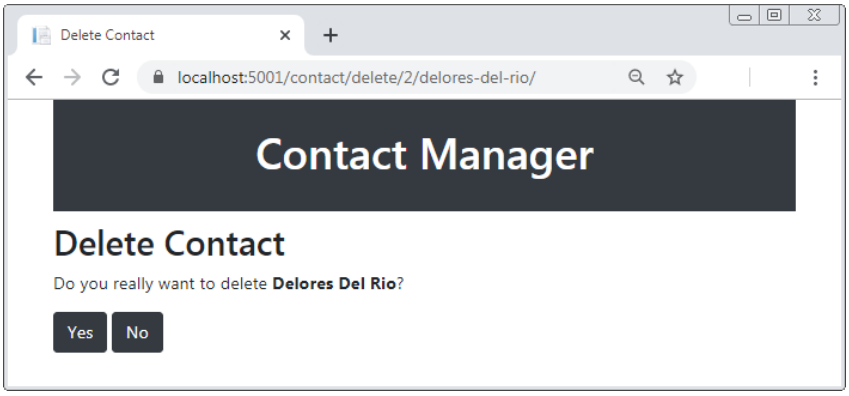
The Details page



The Add and Edit pages (using the same view)



The Delete page



Specifications

- When the application starts, it should display a list of contacts and a link to add a contact
- If the user clicks the first or last name of a contact, the application should display the detail page for that contact
- The Details page should include buttons that allow the user to edit or delete the contact. Before deleting a contact, the application should display the Delete page to confirm the deletion.
- To reduce code duplication, the Add and Edit pages should both use the same view. This view should include a drop-down for Category values.
- The Add and Edit pages should not include the Date Added field that's displayed by the Details page. That field should only be set by code when the user first adds a contact.
- If the user enters invalid data on the Add or Edit page, the application should display a summary of validation errors above the form.
- Here are the requirements for valid data:
    - The Firstname, Lastname, Phone, Email and CategoryId fields are required
    - The Organization field is optional

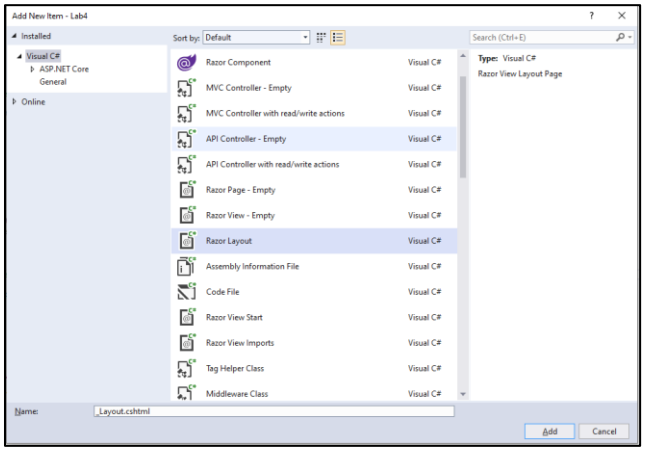
    Note: Since the CategoryId field is likely an int, you can't use the Required validation attribute with it. However, you can use the Range attribute to make sure the value of CategoryId is greater than zero.
- If the user clicks the Cancel button on the Add page, the application should display the Home page
- If the user clicks the Cancel button on the Edit page, the application should display the Details page for that contact.
- The domain model for classes (ex Contacts, Categories) should use primary keys that are generated by the database.
- The Contact class, should have a foreign key field that relates it to the Category class. It should also have a read-only property that creates a slug if the contact's first and lastname that can be added to URLs to make them user friendly.
- Use EF Code first to create a database based on your domain model classes. Include seed data for the categories and one or more contacts.
- Use a Razor layout to store the <html>, <head>, and <body> elements.

- Use the default route (/) with an additional route segment that allows an optional slug at the end of a URL

- Make the application URLs lowercase with trailing slashes

Creating the _Layout.cshtml View

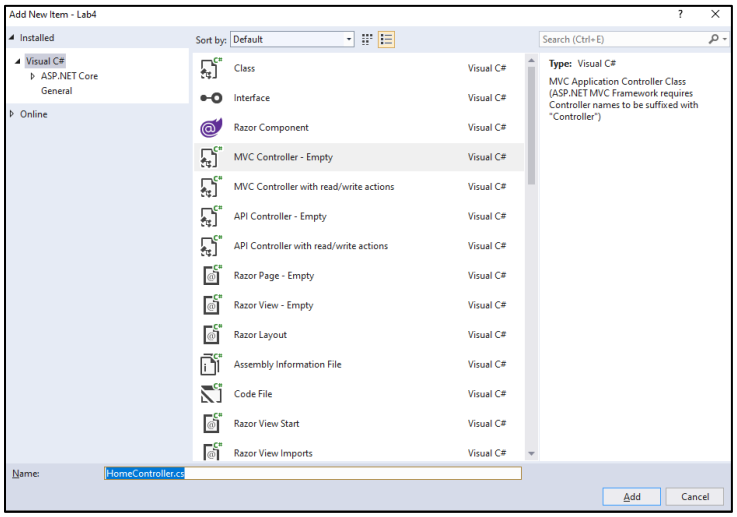
1. Right-click on the **Views→Shared** and create a new Razor Layout view



The **_Layout.cshtml** file created, will act as a template for the application, one that can be used repeatedly across all our views to help create a consistent look and feel, why additionally, reducing the amount of redundancy in our application. Feel free to add any css/javascript etc… here that you want shared across all your application views.

Creating the HomeController

1. Right-click on the Controllers folder, create an Empty controller (Add→Controller→MVC Controller Empty)



2. In the resultant controller stub created in the previous set, we need to add a private **ContactContext** variable (accessor / mutator) to the **HomeController** and add a **HomeController** that accepts a **ContactContext**, to calibrate and assign its value.

```
public class HomeController : Controller
{
    1 reference
    private ContactContext context { get; set; }

    0 references
    public HomeController(ContactContext ctx)
    {
        context = ctx;
    }
```

This code works because of the dependency injection configured in the **Startup.cs**

3. Add a new method in the **HomeController** called **Index**() that returns the list of contacts (query **Contacts** table), include the contact Categories as part of the query, ordering the list by the contacts firstname.

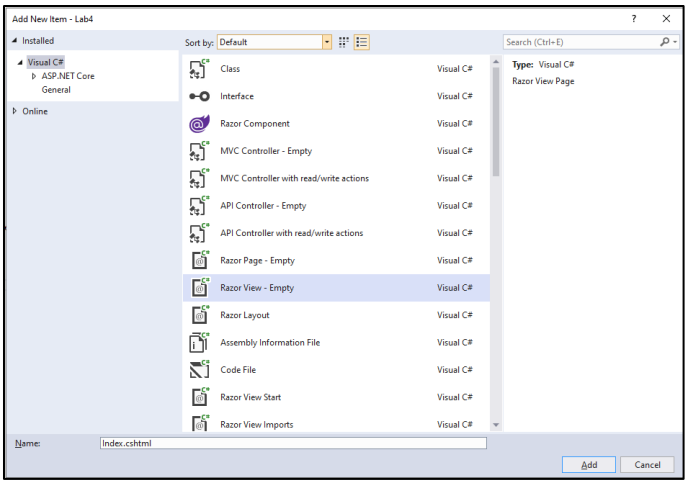
```
public IActionResult Index()
{
    var contacts = context.Contacts
        .Include(c => c.Category)
        .OrderBy(context => context.Firstname).ToList();
    return View(contacts);
}
```

ASP.NET MVC uses a process called "**view discovery**" to match views with controller actions. If you don't pass in a specific view name to look for, it'll look for a view that matches the name of your controller action. So, in this case, it will look for **Index.cshtml** under the View/Home folder.

We will create the view, **Index.cshtml** in the next step.

Creating the Index Page

1. Right-Click on the View/Home folder an create a new Razor (Empty) view (Add →View).



2. In the first line of the view add the @model directive to specify that the model for this view is a **IEnumerable** collection of **Contacts**. The **IEnumerable** make it possible to use the foreach() iteration method in the body of the view.

   Most of the html in this table displays the list of Contacts.

   Within the body of the table, an inline foreach statement loops through the collection of Contact objects ands and displays each one in a new row.

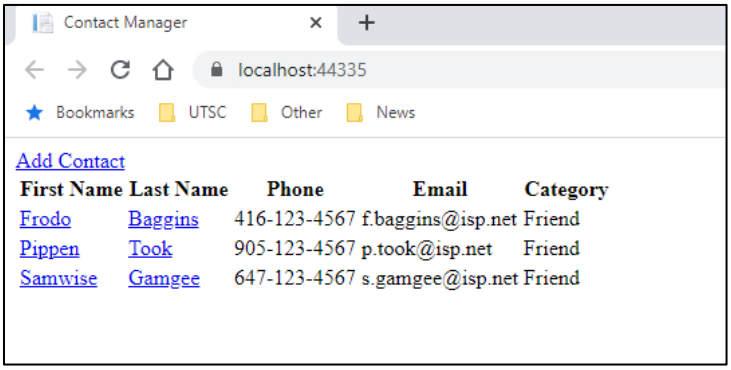
   The following asp directive are also used:

   asp-action → specifies the controller action to invoke
   asp-controller → specifies the controller to invoke
   asp-route-id→helper used to pass the id (primary key) of each contact to the controller
   asp-route-slug→ additional slug parameters to pass to the controller

3. Above the table add a link, that calls the Contact controller (to be implemented in the next section) to add a new contact to the application.

```
<a asp-action="Add" asp-controller="Contact">Add Contact</a>
```

4. Run the application to make sure it renders (as you desire) and without any error(s).



Creating the ContactController

1. Create a second controller called **ContactController** that has:
    a. A private **ContactContext** attribute (setter/getter)
    b. A **constructor** that takes in a **ContactContext** parameter and set the context attribute.

2. Create the following **IActionResult** methods within the **ContactController**

Method Name	GET/POST	Input Parameter
Details()	Get	id
Add()	Get	none
Edit()	Get	id
Edit()	Post	Contact
Delete()	Get	id
Delete()	Post	Contact

3. Implement the Details() method that accepts a contact id (int), and forwards the contact associated with the id to the Details view (**Details.cshtml** – to be completed)

```
public IActionResult Details(int id)
{
    var contact = context.Contacts
        .Include(c => c.Category)
        .FirstOrDefault(c => c.ContactId == id);
    return View(contact);
}
```

4. Implement the **Add**() method that forwards a **new Contact()** to the "Edit" view (the add and edit view utilize the same view). The Add() method also should set an **Action** property and **Categories** property in the **ViewBag** of the Model.

```
public IActionResult Add()
{
    ViewBag.Action = "Add";
    ViewBag.Categories = context.Categories.OrderBy(c => c.Name).ToList();

    return View("Edit", new Contact());
}
```

5. Implement the **Edit**() that accepts the id of the contact to edit, the method then forwards the **Contact** to be edited to the "Edit" view. This **Edit**() method also should set an **Action** property and **Categories** property in the **ViewBag** of the Model.

```
public IActionResult Edit(int id)
{
    ViewBag.Action = "Edit";
    ViewBag.Categories = context.Categories.OrderBy(c => c.Name).ToList();

    var contact = context.Contacts
        .Include(c => c.Category)
        .FirstOrDefault(c => c.ContactId == id);
    return View(contact);
}
```

6. Implement the **Edit**() method that accepts a **Contact** object that is to be edited. This **Edit**() method should first validate that the data submitted is valid.

    - Retrieve the contact Id of the contact passed if the value is > 0, then the action is to **Edit**, otherwise the action is to **Add**.
    - If the action set is to **Add**, an contact passed, should be added (persisted) to the database via a call to **context.Contact.Add()** with the date timestamp first added.
    - Otherwise if the action set is to Edit, then a call to **context.Contacts.Update()** should be called since we do not want to persist a duplicate contact, as we just want to update an already known contact.
    - To persist changes (add, or delete) we must make a call to **context.SaveChanges()**.
    - If we have successfully persisted a new or updated contact we should issue a redirect (**RedirectToAction**()) to send the user back to the index (home page).
    - If any of the contact data passed is missing or not valid, we should set the action and category in the Model **ViewBag**, back to their passed value, then forward back to the Edit view.

```
public IActionResult Edit(Contact contact)
{

    string action = (contact.ContactId == 0) ? "Add" : "Edit";

    if(ModelState.IsValid)
    {
        if(action == "Add")
        {
            contact.DateAdded = DateTime.Now;
            context.Contacts.Add(contact);
        }else
        {
            context.Contacts.Update(contact);
        }
        context.SaveChanges();

        return RedirectToAction("Index", "Home");
    }
    else
    {
        ViewBag.Action = action;
        ViewBag.Categories = context.Categories.OrderBy(c => c.Name).ToList();
        return View(contact);
    }

}
```

7. Implement the **Delete**() method that accepts the **id** of the contact to delete and forwards to the Delete view (**Delete.cshtml** – to be implemented)

```
public IActionResult Delete(int id)
{
    var contact = context.Contacts
        .Include(c => c.ContactId == id)
        .FirstOrDefault();
    return View(contact);
}
```

8. Implement the **Delete**() method that accepts the **contact** object to delete and persists the changes, then redirects back to the home index page.

```
public IActionResult Delete(Contact contact)
{
    context.Contacts.Remove(contact);
    context.SaveChanges();
    return RedirectToAction("Index", "Home");
}
```

9. For each of the methods, verify that you have associated them with the appropriate **HttpGet** or **HttpPost** as specified in the requirements.

Questions

1. On your own create the following remaining views in the Views/Contact directory of your project:
    i. Delete.cshtml
    ii. Details.cshtml
    iii. Edit.cshtml

   Make sure to implement the view so they match the wireframe provided above.

2. EXTRA → Though look and feel is not theme or this course or chapter, try to implement some styling (css, boostrap etc..) to liven up the user experience of the application.

   *** Make sure to commit your changes if you created a GitHub account ***