ASP.NET MVC Introduction

Estimated time for completion: 30-60 minutes

Overview:

In this lab you will use ASP.NET MVC to view album data from the database. You will also build a page that will accept updates for the album data.

Goals:

- Define a controller to accept requests
- Implement a view page to show results from controller actions
- Implement a view page to accept user inputs
- Accept user inputs to controller actions

Simple controller and view

You will define a controller that receives requests for a listing of albums, access album model data and pass that data to a view for display.

Criteria:

- Define a album controller and Index controller action to access album data
- Implement view page to display album list

Steps:

1. Open the solution from `~/MVCIntro/before`.
    a. Inspect `default.aspx`. The default page is in place for IIS' (and Cassini's) default document mapping mechanism and notice it simply has a link to the album index page (which we have yet to implement).
2. We now need to define the album controller. This will manage all requests for album related operations. For now, the only request we will implement is for a listing of all albums, so this will be the `Index` action.
    a. Add a new `AlbumController` class to the `Controllers` directory by right-clicking the directory and choose the Add → Controller menu option. Other than setting the name, leave all the other settings at their default values.
    b. Notice an `Index` method is automatically added. This will be the first action you implement. Also notice it simply returns `View()`. Without any modifications we

have an action method that returns a view engine result. We will implement some logic later to load album data.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace WebApp.Controllers
{
    public class AlbumController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

3. You will now define the view page for the `Index` action.
    a. In the `Views` directory add a new child directory called `Album`. This will contain the view templates for rendering HTML.
    b. Add a new `Index` view by right clicking the `Views/Album` directory and choosing Add → View. Uncheck all the options on the "Add View" dialog.
    c. For now in `Index.cshtml` simply add a `<h1>` that says `"Album Index"`.
4. Run your application to ensure you can access the `Album`'s `Index` action and it renders your `Index` view template. The URL in the browser should be `~/Album/Index`.
5. Now that we can see the controller and view being accessed, we need to go back and properly implement the `Index` action to access the database and display some useful information to the user. This means we need a way to access the database. We will use the Chinook project that's already in the solution.

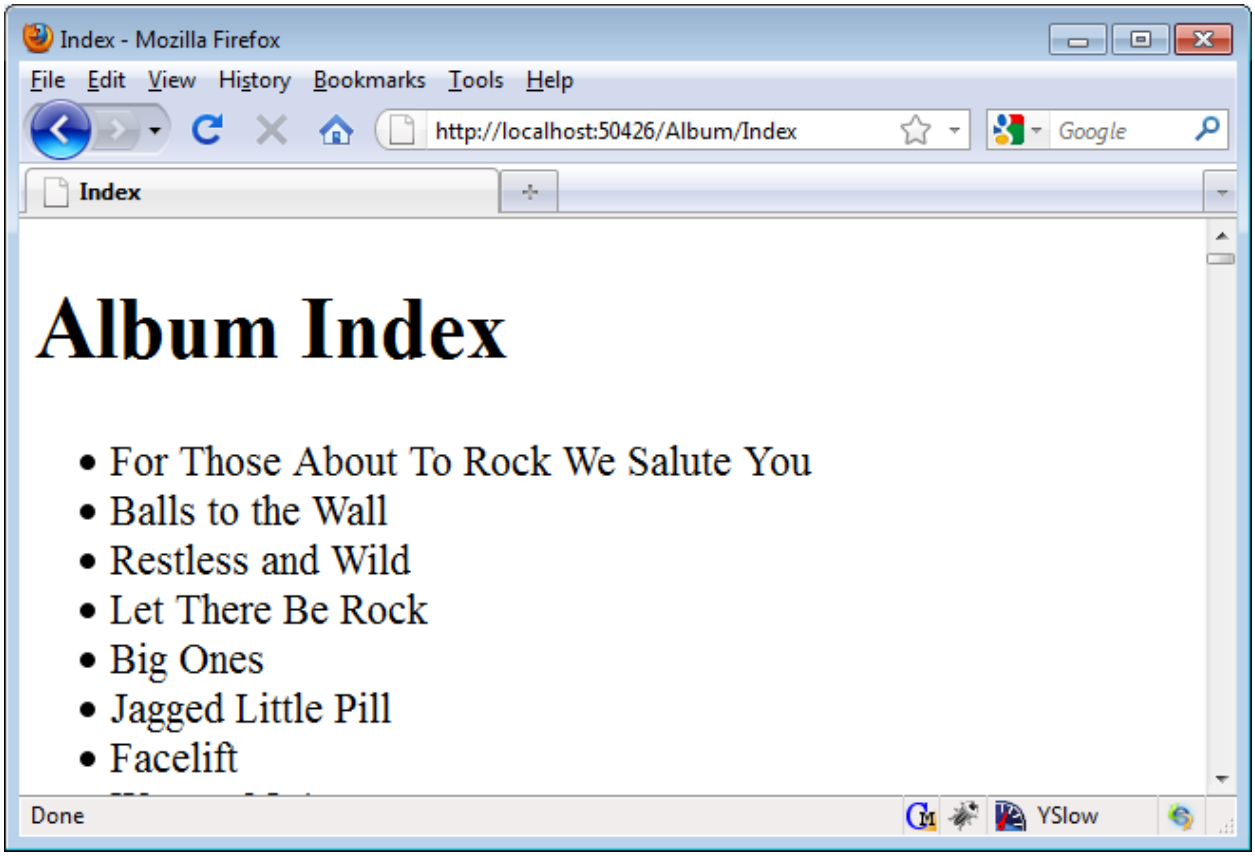
    In the `AlbumController`'s `Index` action method, use the `SqlCompactMusicRepository` class to access the database and pass it to the view:

    a. Create an instance of the `SqlCompactMusicRepository` and invoke the `GetAlbums` API. This is the data we need to pass to the view for display. You might need a `using` statement for the `Chinook.Repository.SqlCompact` namespace.
    b. Use the `ViewData` dictionary to pass the album list to the view. Assign the album list into an entry in the dictionary (perhaps using `"Albums"` as the dictionary key). This data will now be available in the view.

c. The method should already be returning `View()` so there is nothing else to do here.

```
public class AlbumController : Controller
{
    public ActionResult Index()
    {
        var musicRepository = new SqlCompactMusicRepository();
        ViewData["Albums"] = musicRepository.GetAlbums();
        return View();
    }
}
```

6. We now need to modify the `Index` view to show the list of albums for the album controller. It's supposed to look something like:



a. Open `Views\Album\Index.cshtml`.

b. After the `<h1>` add a `<ul>` which will contain the list of albums.

c. Inside the `<ul>` add a `foreach` iterating over each album in the list of albums (which is accessible in the `ViewData["Albums"]` as an `IEnumerable<Chinook.DomainModel.Album>`).

d. Inside the `foreach` emit a `<li>` that contains the album's title.

```
<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    <h1>Album Index</h1>
    <ul>
        @foreach (var album in
           (IEnumerable<Chinook.DomainModel.Album>)ViewData["Albums"])
        {
            <li>
                @album.Title
            </li>
        }
    </ul>
</body>
</html>
```

7. You should now be able to test your page and see the list of albums in the browser.

Adding an edit action and view

You will define an action and view for editing album information.

Criteria:

- Implement a view that allows users to enter data
- Use Html helpers to assist in rendering HTML
- Define an action that accepts input parameters

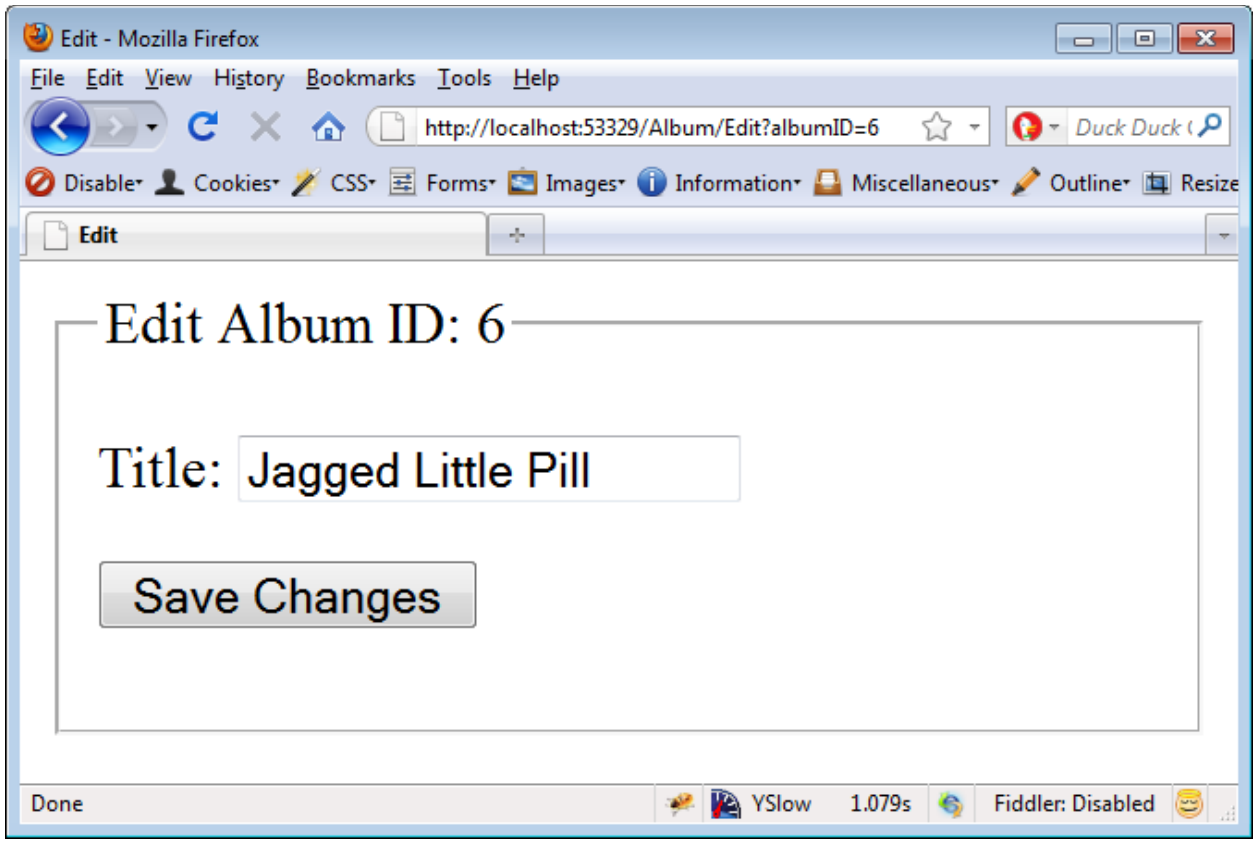
Steps:

1. In this part of the lab you will be adding the capability for a user to navigate to an album edit page, allowing them to change the title of an album.
2. A user will make a new request to view the album details. The URL will look something like `~/Album/Edit?albumID=5`. This will require a new action on the `AlbumController`.
    a. Add a new action method called `Edit`. It should accept an integer parameter called `albumID`. This parameter name is required since it will be the name of the query string parameter.
    b. Create a new instance of the `SqlCompactMusicRepository` class. Use the `GetAlbum` API passing in the `albumID` parameter to access the album requested. The return is an `Album` object which will be passed to the view for display.

c. For the `Edit` action method return value and thus the view to display to the user, use the `View` API. To pass the album object you will pass it as a strongly typed object, thus it must be assign to the `Model` property or passed as the parameter to `View()`.

```
public ActionResult Edit(int albumID)
{
    var repository = new SqlCompactMusicRepository();
    var album = repository.GetAlbum(albumID);
    return View(album);
}
```

3. You now need to create the view to display the album to the user. It is supposed to look something like:



a. Add a new `Edit` view for the album controller.

- Do not choose a master page.
- Make it strongly typed on `Chinook.DomainModel.Album`

Notice at the top of the new created `Edit.cshtml` the `@model` directive was set for you. This is what makes this view template strongly typed for the `Album` class.

b. Under the `<body>` add a `<fieldset>`.

c. Inside the `<fieldset>` add a `<legend>` to display some text to describe which album ID is being displayed. You can access the album object from the `Model`.

d. After the `<legend>`, add a `<p>` tag. This will contain a text box (`<input type=text>`) for the album title. Use the `Html.TextBox` helper API to render a text box.

e. Add another `<p>` tag. Inside add a submit button (`<input type='submit' value='Save Changes'>`).

```
@model Chinook.DomainModel.Album

<!DOCTYPE html>

<html>
<head>
    <title>Edit</title>
</head>
<body>
  <fieldset>
      <legend>Edit Album ID: @Model.AlbumID</legend>
      <p>
          <label for="title">Title:</label>
          @Html.TextBox("title", Model.Title)
      </p>
      <p>
          <input type="submit" value="Save Changes" />
      </p>
  </fieldset>
</body>
</html>
```

We've yet to add a `<form>` but for now this is sufficient.

4. Run you page to test that the album information is being loaded into the page. You might need to manually type the URL into the browser since the album listing page doesn't currently link to the show action. The URL will look something like `/Album/Edit?albumID=5`.

5. Let's fix the problem where the album listing does not provide links to the details page.

   a. Open the `Index` view page.

   b. Remove the code inside the `<li>`. Replace it with a hyperlink to link to the `Edit` action. Use the `Html.ActionLink` helper API for this. You will need to pass as parameters:

      - the text for the link (which will be the album title)
      - the name of the action to link to (which will be `"Edit"`)

- an anonymous object that contains the parameter to pass to the action (the parameter name is `albumID` and must be assigned from the album's `AlbumID` property)

```
@Html.ActionLink(album.Title, "Edit", new { albumID = album.AlbumID })
```

6. Run the index page again to test that the links now allow navigation to the show action.

Complete the update code

You will create an action method to accept the updates for the album information.

Criteria:

- Define actions that accept parameters
- Add a form to the edit page

Steps:

1. In this part of the lab you will be completing the edit view and adding a new action method to accept the album updates.
2. First add a new action method to the album controller called `Update`.
    a. It should accept two parameters: an `int` parameter called `albumID` and a `string` parameter called `title`.
    b. Inside the method create a new instance of the `SqlCompactMusicRepository` class.
    c. Use the `albumID` parameter to load an instance of the `Album` from the repository via the `GetAlbum` API.
    d. Assign the `Title` property to the album object from the `title` parameter.
    e. Save the album object back to the repository via the `SaveAlbum` API.

```
public ActionResult Update(int albumID, string title)
{
    var musicRepository = new SqlCompactMusicRepository();
    var album = musicRepository.GetAlbum(albumID);
    album.Title = title;
    musicRepository.SaveAlbum(album);

    return RedirectToAction("Index");
}
```

3. Now we need to update the edit view to post to the update action method.
    a. Open `Views\Album\Edit.cshtml`.

b. Wrap a `<form>` around the `<fieldset>` by using the `Html.BeginForm` helper API.

- Pass `"Update"` for the action parameter
- Pass `"Album"` for the controller parameter

c. We need to pass the album ID. Inside the `<fieldset>` add a `<input type=hidden>` by using the `Html.Hidden` helper API.

- Pass `"albumID"` for the name parameter
- Pass `Model.AlbumID` for the value parameter

```
@using (Html.BeginForm("Update", "Album"))
{
    <fieldset>
        <legend>Edit Album ID: @Model.AlbumID</legend>
        @Html.Hidden("albumID", Model.AlbumID)
        <p>
            <label for="title">Title:</label>
            @Html.TextBox("title", Model.Title)
        </p>
        <p>
            <input type="submit" value="Save Changes" />
        </p>
    </fieldset>
}
```

4. Test that the edit page now allows a user to update an album title.

[OPTIONAL] Adding paging support

On the album index page you will add support for paging.

Criteria:

- Add paging support to the Index action and view

Steps:

To support paging in the Index action, you will accept input to indicate the page number to display. A query string parameter will be used to indicate the page number to display.

a. Open the source file for the `AlbumController` class and add an `int?` parameter for the requested page number. We are using a nullable for the parameter in case the user does not pass any input at all.

b. Use the page number parameter to calculate the starting row number that needs to be retrieved from the repository. We will be using a page size of 5.

c. Use the `Skip` and `Take` LINQ APIs on the return value from `GetAlbums` to access only the rows necessary to display the page being requested.

d. Assign the returned list of albums to `ViewData["Albums"]` as was being done before.

```
public ActionResult Index(int? page)
{
    int pageSize = 5;
    int currentPage = page ?? 1;
    if (currentPage < 1) currentPage = 1;
    int startRow = (currentPage - 1) * pageSize;

    var repository = new SqlCompactMusicRepository();
    ViewData["Albums"] =
       repository.GetAlbums()
                 .OrderBy(x=>x.AlbumID)
                 .Skip(startRow)
                 .Take(pageSize);
    return View();
}
```

1. Run and test `/Album/Index?page=1` to see if now only 5 albums are listed. Pass other page numbers to the query string to see if other pages of data are properly shown.
2. If you're feeling adventurous, add support for "First", "Last", "Next" and "Prev" links in the view to allow the user to click to access other pages of data.

   This will involve calculating the total number of pages possible. Also, it might be necessary to pass additional data to the view to support this feature. To pass more data to the view use the `ViewData` dictionary or even create a view model style class to hold all the necessary data.

Solutions:

The final solution for this lab is available in the `~/after` directory