Course: Development of Business Applications  
Faculty of Economics in Osijek, University of Osijek

Laboratory exercises

Lab: Developing ASP.NET MVC 4 Movie Web Application

Introduction


Through the following exerceses you will implement a simple movie-listing application that supports creating, editing, searching and listing movies from a database. The application also lets you add, edit, and delete movies, as well as see details about individual ones. All data-entry scenarios include validation to ensure that the data stored in the database is correct.

Exercise 1: Creating an MVC Project and Adding an MVC Controller

After completing Exercise 1, you will be able to:

- Add an MVC controller to a web application.
- Write actions in an MVC controller that respond to user operations such as create, index, display, and delete.
- Write action filters that run code for multiple actions.

Scenario

In this exercise, you will:

- Create a new MVC 4 web application in Visual Studio 2012.
- Add MVC controller to the web application.

The main tasks for this exercise are as follows:

1. Create a new MVC project.
2. Add a new MVC controller.

Task 1: Create a new MVC project
1. Start Visual Studio 2012 and create a new ASP.NET MVC 4 Web Application

![New project in Visual Studio 2012](image)

*Picture 1: New project in Visual studio 2012*

2. Use the following information:
   - Name: MvcMovie
   - Location: Allfiles (D):\Labfiles\ 
   - Solution name: MvcMovie
   - Create directory for solution: True
   - Project Template: ASP.NET MVC 4 Web Application
3. In the New ASP.NET MVC 4 Project dialog box, select Internet Application. Leave Razor as the default view engine.
4. Click OK.
5. Run the application by debugging (Debug menu, select Start Debugging – F5).

Right out of the box this default template gives you Home, Contact and About pages. It also provides support to register and log in, and links to Facebook and Twitter. The next step is to add new Controller.

Task 2: Add a new MVC Controller

1. In Solution Explorer, right-click the Controllers folder and then select Add Controller.
2. Name your new controller "HelloWorldController". Leave the default template as Empty MVC controller and click Add.
Notice in Solution Explorer that a new file has been created named HelloWorldController.cs. The file is open in the IDE.

*Picture 5: Empty MVC controller*
3. Replace the contents of the file with the following code:
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        // GET: /HelloWorld/
        public string Index()
        {
            return "This is my <b>default</b> action...";
        }

        // GET: /HelloWorld/Welcome/
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}

4. The controller methods will return a string of HTML as an example. The controller is named HelloWorldController and the first method above is named Index. Let’s invoke it from a browser. Run the application by debugging, and access the following relative path:

• /HelloWorld

For example, in the illustration below, it's http://localhost:1234/HelloWorld.
6. The Welcome method runs and returns the string "This is the Welcome action method...".

Picture 7: Running Index action from HelloWorld controller in browser

```
This is my default action...
```

Picture 8: Welcome method output in browser

```
This is the Welcome action method...
```
7. The default MVC mapping is /[Controller]/[ActionName]/[Parameters]. For this URL, the controller is HelloWorld and Welcome is the action method. You haven't used the [Parameters] part of the URL yet.

8. Modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, /HelloWorld/Welcome?name=Scott&numtimes=4).

9. Change your Welcome method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the numTimes parameter should default to 1 if no value is passed for that parameter.

```csharp
public string Welcome(string name, int numTimes = 1) {
    return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```

10. Run your application and browse to the example URL (http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4). You can try different values for name and numtimes in the URL. The ASP.NET MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method.

   ![Running modified Welcome method](image.png)

   *Picture 9: Running modified Welcome method*

   In both these examples the controller has been doing the "VC" portion of MVC — that is, the view and controller work. The controller is returning HTML directly. Ordinarily you don’t want controllers returning HTML directly, since that becomes very cumbersome to code. Instead we'll typically use a separate view template file to help generate the HTML response.
Exercise 2: Adding a View

Scenario
After completing Exercise 2, you will be able to:

- Add Razor views to an MVC application and set properties such as scaffold and model binding.
- Write both HTML markup and C# code in a view by using Razor syntax.

The main tasks for this exercise are as follows:

1. Add a new display view and you’re going to modify the HelloWorldController class to use view template files.
2. Changing Views and Layout Pages
3. Passing Data from the Controller to the View

Task 1: Add a new display view

1. Currently the Index method returns a string with a message that is hard-coded in the controller class. Change the Index method to return a View object, as shown in the following code:

```csharp
public ActionResult Index()
{
    return View();
}
```

2. The Index method above uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods), such as the Index method above, generally return an ActionResult (or a class derived from ActionResult), not primitive types like string.
3. Add a view template that you can use with the Index method. To do this, right-click inside the Index method and click Add View.

```csharp
public class HelloWorldController : Controller
{
    // GET: /HelloWorld/

    public ActionResult Index()
    {
        return View();
    }

    // GET: /HelloWorld/
    public string Welcome()
    {
        return "Hello, World!";
    }
}
```
4. The **Add View** dialog box appears. Leave the defaults the way they are and click the **Add** button:

![Add View dialog box](image11.png)

*Picture 11: Add View dialog box*

5. The MvcMovie\Views\HelloWorld folder and the MvcMovie\Views\HelloWorld\Index.cshtml file are created. You can see them in **Solution Explorer**: 

![Solution Explorer](image12.png)
6. The following shows the Index.cshtml file that was created:
7. Add the following HTML under the <h2> tag:
<p>Hello from our View Template!</p>

8. The complete MvcMovie\Views\HelloWorld\Index.cshtml file is shown below:
```csharp
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>Hello from our View Template!</p>
```

10. The image below shows the string "Hello from our View Template!" hard-coded in the view:
Task 2: Changing Views and Layout Pages

1. First, you want to change the "your logo here." title at the top of the page. That text is common to every page. It's actually implemented in only one place in the project, even though it appears on every page in the application. Go to the /Views/Shared folder in Solution Explorer and open the _Layout.cshtml file. This file is called a layout page and it's the shared "shell" that all other pages use.
Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. For example, if you select the About link, the `Views\Home\About.cshtml` view is rendered inside the `RenderBody` method.

2. Change the site-title heading in the layout template from "your logo here" to "MVC Movie":

Picture 16: _Layout.cshtml file
3. Replace the contents of the title element with the following markup:

```
<title>@ViewBag.Title - Movie App</title>
```

4. Run the application and notice that it now says "MVC Movie". Click the About link, and you see how that page shows "MVC Movie", too. We were able to make the change once in the layout template and have all pages on the site reflect the new title:

![About - Movie App](image)

```
About. Your app description page.
```

5. Change the title of the Index view. Open MvcMovie\Views\HelloWorld\Index.cshtml. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). You’ll make them slightly different so you can see which bit of code changes which part of the app.

![Debugging MvcMovie Application](image)

```
© 2012 - My ASP.NET MVC Application
```
To indicate the HTML title to display, the code above sets a Title property of the ViewBag object (which is in the Index.cshtml view template). If you look back at the source code of the layout template, you’ll notice that the template uses this value in the <title> element as part of the <head> section of the HTML that we modified previously. The browser title is created with the ViewBag.Title we set in the Index.cshtml view template and the additional “- Movie App” added in the layout file.

Also notice how the content in the Index.cshtml view template was merged with the _Layout.cshtml view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application.
Task 3: Passing Data from the Controller to the View

1. Return to the HelloWorldController.cs file and change the Welcome method to add a Message and NumTimes value to the ViewBag object. ViewBag is a dynamic object, which means you can put whatever you want in to it; the ViewBag object has no defined properties until you put something inside it. The ASP.NET MVC model binding system automatically maps the named parameters (name and numTimes) from the query string in the address bar to parameters in your method. The complete HelloWorldController.cs file looks like this:

```csharp
using System;
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;

            return View();
        }
    }
}
```

Now the ViewBag object contains data that will be passed to the view automatically.

Next, you need a Welcome view template! In the Build menu, select Build MvcMovie to make sure the project is compiled.

2. Then right-click inside the Welcome method and click Add View.
Picture 19: Adding a View
3. **Add View** dialog box looks like:

![Add View dialog box](image)

**Picture 20: Add View dialog box**

4. Click **Add**, and then add the following code under the `<h2>` element in the new `Welcome.cshtml` file. You'll create a loop that says "Hello" as many times as the user says it should. The complete `Welcome.cshtml` file is shown below.

```html
@{
    ViewBag.Title = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i=0; i < ViewBag.NumTimes; i++) {
        <li>@ViewBag.Message</li>
    }
</ul>
```
5. Run the application and browse to the following relative path:

http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4

Now data is taken from the URL and passed to the controller using the model binder. The controller packages the data into a ViewBag object and passes that object to the view. The view then displays the data as HTML to the user.

Exercise 3: Adding a Model

Scenario
After completing Exercise 3, you will be able to:

- Add a new model to the ASP.NET MVC 4 web application and add properties to the model.
- Use Visual Studio to create a new SQL LocalDB database and connect to the database.
- Add Entity Framework code to the model classes in the MVC model.

The main tasks for this exercise are as follows:

1. Add a new MVC model
2. Creating a Connection String and Working with SQL Server LocalDB
Task 1: Add a new MVC model

1. **In Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**:

   ![Solution Explorer](image)

   - Controller...
   - New Item...
   - Existing Item...
   - New Folder
   - Add ASP.NET Folder
   - Class...

   ![Class](image)

   ![Add](image)

   ![Scope to This](image)

   ![New Solution Explorer View](image)

   ![Cut](image)

   ![Copy](image)

   ![Paste](image)

   ![Delete](image)

   ![Rename](image)

   ![Open Folder in File Explorer](image)

   ![Properties](image)

   ![Alt+Enter](image)

   **Picture 22: Adding new class in MVC project**

2. Enter the class name "Movie".
3. Add the following five properties to the Movie class:

   ```csharp
   public class Movie
   {
       public int ID { get; set; }
       public string Title { get; set; }
       public DateTime ReleaseDate { get; set; }
       public string Genre { get; set; }
       public decimal Price { get; set; }
   }
   ```
4. We'll use the Movie class to represent movies in a database. Each instance of a Movie object will correspond to a row within a database table, and each property of the Movie class will map to a column in the table. In the same file, add the following MovieDBContext class:

```csharp
public class MovieDBContext : DbContext
{
    public DbSet<Movie> Movies { get; set; }
}
```

5. The MovieDBContext class represents the Entity Framework movie database context, which handles fetching, storing, and updating Movie class instances in a database. The MovieDBContext derives from the DbContext base class provided by the Entity Framework. In order to be able to reference DbContext and DbSet, you need to add the following using statement at the top of the file:

```csharp
using System.Data.Entity;
```

6. The complete Movie.cs file is shown below. (Several using statements that are not needed have been removed.):

```csharp
using System;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        ... } 

    public class MovieDBContext : DbContext
    { 
        public DbSet<Movie> Movies { get; set; } 
    } 
}
```

**Task 2: Creating a Connection String and Working with SQL Server LocalDB**

1. The MovieDBContext class you created handles the task of connecting to the database and mapping Movie objects to database records. One question you might ask, though, is how to specify which database it will connect to. You'll do that by adding connection information in the Web.config file of the application. Open the application root Web.config file. (Not the Web.config file in the Views folder.) Open the Web.config file outlined in red:
2. Add the following connection string to the `<connectionStrings>` element in the `Web.config` file.

```xml
<add name="MovieDBContext" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated Security=True" providerName="System.Data.SqlClient" />
```

3. The following example shows a portion of the `Web.config` file with the new connection string added:

```xml
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-MvcMovie-2012213181139;Integrated Security=true" providerName="System.Data.SqlClient"/>
  <add name="MovieDBContext" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated Security=True" providerName="System.Data.SqlClient"/>
</connectionStrings>
```
Exercise 4: Accessing Your Model's Data from a Controller

Scenario
After completing Exercise 4, you will be able to:

- Add new Controller class and write code that retrieves the Model's data
- Display data in the browser using a View template
- Adding a Search Method

The main tasks for this exercise are as follows:

1. Creating a new MoviesController class and write code that retrieves the movie data and displays it in the browser using a view template.
2. Working with SQL Server LocalDB.
3. Creating and displaying SearchIndex Form
4. Adding Search by Genre

Task 1: Creating a new MoviesController class and write code that retrieves the movie data and displays it in the browser using a view template

1. **Build the application** before going on to the next step.
2. Right-click the Controllers folder and create a new MoviesController controller. The options below will not appear until you build your application. Select the following options:
   - Controller name: MoviesController. (This is the default.)
   - Template: MVC Controller with read/write actions and views, using Entity Framework.
   - Model class: Movie (MvcMovie.Models).
   - Data context class: MovieDBContext (MvcMovie.Models).
   - Views: Razor (CSHTML). (The default.)
3. Click **Add**. Visual Studio Express creates the following files and folders:
   - A `MoviesController.cs` file in the project's Controllers folder.
   - A Movies folder in the project's Views folder.
   - Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml, and Index.cshtml in the new Views\Movies folder.

   ASP.NET MVC 4 automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as scaffolding). You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

   Run the application and browse to the Movies controller by appending `/Movies` to the URL in the address bar of your browser. Create some Movies and examine **Edit**, **Details**, and **Delete** links.
Task 2: Working with SQL Server LocalDB

1. Entity Framework Code First detected that the database connection string that was provided pointed to a Movies database that didn’t exist yet, so Code First created the database automatically. You can verify that it’s been created by looking in the App_Data folder. If you don’t see the Movies.mdf file, click the Show All Files button in the Solution Explorer toolbar, click the Refresh button, and then expand the App_Data folder.
2. Double-click Movies.mdf to open DATABASE EXPLORER, then expand the Tables folder to see the Movies table.

3. Right-click the Movies table and select Show Table Data to see the data you created.
4. Right-click the Movies table and select **Open Table Definition** to see the table structure that Entity Framework Code First created for you.
Notice how the schema of the Movies table maps to the Movie class you created earlier. Entity Framework Code First automatically created this schema for you based on your Movie class.

5. When you're finished, close the connection by right clicking MovieDBContext and selecting Close Connection. (If you don't close the connection, you might get an error the next time you run the project).
Task 3: Creating and displaying SearchIndex Form

1. **Adding a SearchIndex action method to the existing MoviesController class.** The method will return a view that contains an HTML form. Here's the code:
public ActionResult SearchIndex(string searchString)  
{            
    var movies = from m in db.Movies  
           ...    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }  
  
    return View(movies);  
}

The first line of the SearchIndex method creates the following LINQ query to select the movies:

```csharp
var movies = from m in db.Movies  
           select m;
```

The query is defined at this point, but hasn't yet been run against the data store. If the searchString parameter contains a string, the movies query is modified to filter on the value of the search string, using the following code:

```csharp
if (!String.IsNullOrEmpty(searchString))  
    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }
```

2. Implement the `SearchIndex` view that will display the form to the user. Right-click inside the `SearchIndex` method and then click `Add View`. In the `Add View` dialog box, specify that you're going to pass a `Movie` object to the view template as its model class. In the `Scaffold template list`, choose `List`, then click `Add`. 


3. Click the **Add** button, the `Views\Movies\SearchIndex.cshtml` view template is created. Because you selected **List** in the **Scaffold template** list, Visual Studio automatically generated (scaffolded) some default markup in the view. The scaffolding created an HTML form. It examined the `Movie` class and created code to render `<label>` elements for each property of the class. The listing below shows the Create view that was generated:
@model IEnumerable<MvcMovie.Models.Movie>

{@
    ViewBag.Title = "SearchIndex";
}

<h2>SearchIndex</h2>

<p>@Html.ActionLink("Create New", "Create")</p>

<table>
  <tr>
    <th>Title</th>
    <th>ReleaseDate</th>
    <th>Genre</th>
    <th>Price</th>
    <th></th>
  </tr>
  @foreach (var item in Model) {
    <tr>
      <td>@Html.DisplayFor(modelItem => item.Title)</td>
      <td>@Html.DisplayFor(modelItem => item.ReleaseDate)</td>
      <td>@Html.DisplayFor(modelItem => item.Genre)</td>
      <td>@Html.DisplayFor(modelItem => item.Price)</td>
      <td>@Html.ActionLink("Edit", "Edit", new { id=item.ID }) | @Html.ActionLink("Details", "Details", new { id=item.ID }) | @Html.ActionLink("Delete", "Delete", new { id=item.ID })</td>
    </tr>
  }
</table>

4. Run the application and navigate to /Movies/SearchIndex. Append a query string such as ?searchString=ghost to the URL. The filtered movies are displayed.
5. Open the Views\Movies\SearchIndex.cshtml file, and just after @Html.ActionLink("Create New", "Create"), add the following:

@using (Html.BeginForm("SearchIndex","Movies",FormMethod.Get))
{
    <p>
    Title: @Html.TextBox("SearchString")
    <input type="submit" value="Filter" /></p>
}

The following example shows a portion of the Views\Movies\SearchIndex.cshtml file with the added filtering markup.
The `Html.BeginForm` helper creates an opening `<form>` tag. The `Html.BeginForm` helper causes the form to post to itself when the user submits the form by clicking the `Filter` button.

6. Run the application and try searching for a movie.

**Task 4: Adding Search by Genre**

1. Add a feature to let users search for movies by genre. Replace the `SearchIndex` method with the following code:

```csharp
public ActionResult SearchIndex(string movieGenre, string searchString)  
{  
    var GenreLst = new List<string>();  
    var GenreQry = from d in db.Movies  
                    orderby d.Genre  
                    select d.Genre;  
    GenreLst.AddRange(GenreQry.Distinct());  
    ViewBag.movieGenre = new SelectList(GenreLst);  
    var movies = from m in db.Movies  
                 select m;  
    if (!String.IsNullOrEmpty(searchString))  
    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }  
    if (string.IsNullOrEmpty(movieGenre))  
        return View(movies);  
    else  
    {  
        return View(movies.Where(x => x.Genre == movieGenre));  
    }  
}  
```
2. Add an Html.DropDownList helper to the Views\Movies\SearchIndex.cshtml file, just before the TextBox helper. The completed markup is shown below:

```html
@Html.ActionLink("Create New", "Create")
@using (Html.BeginForm("SearchIndex","Movies",FormMethod.Get)){
    Genre: @Html.DropDownList("movieGenre", "All")
    Title: @Html.TextBox("SearchString")
    <input type="submit" value="Filter" /></p>
}
</p>
```

3. Run the application and browse to /Movies/SearchIndex. Try a search by genre, by movie name, and by both criteria.

![SearchIndex](image-url)

*Picture 33: Filtered movies by genre and title*

References:

2. Schoolfreeware, Visual Studio Express 2012 For Web Tutorials (2012),

3. Pluralsight, Building Applications with ASP.NET MVC 4 (2012),
http://pluralsight.com/training/Player?author=scott-allen&name=mvc4-building-m1-intro&mode=live&clip=0&course=mvc4-building, [20.10.2013.]