

ASP.NET MVC 5

About this Lab Manual

This lab manual provides a series of hands-on exercises for learning how to build web applications using Microsoft's ASP.NET MVC 5, C#, and Visual Studio 2015.

Conventions

Each hands-on exercise in this manual will consist of a series of steps to accomplish a learning objective. Additional information will be presented using the following icons:



General information



Tips and tricks



What needs to be done



What not to do



What you've accomplished



Performance advice



How things work and why



Maintenance advice



Where to find more information

Copyright © 2014-2016

Treeloop, LLC

All rights reserved.

No part of this book may be reproduced in any form or by any electronic or mechanical means including information storage and retrieval systems, without permission in writing from the author.

Requirements

- **Visual Studio 2013 (any edition other than Express) or Visual Studio 2015 (any edition)**
 - If using Visual Studio 2013, you should have **Update 5** (or later) installed. You can check the update level installed by selecting [**Help > About...**] within Visual Studio. Update 5 can be downloaded from <<https://www.microsoft.com/en-us/download/details.aspx?id=48129>>
 - **LocalDB or SQL Server (any version)**
 - Installed by default as part of the Visual Studio installation process
 - To confirm the installation of LocalDB, open Visual Studio, select [**View > SQL Server Object Explorer**] from the menu, click the button to **Add a SQL Server**, and use **(localdb)/v11.0** as the server name. If LocalDB is installed, you should be able to successfully create the connection
 - If not installed, you can download an installer for LocalDB from <<http://www.microsoft.com/en-us/download/details.aspx?id=29062>> Choose the file named **SqlLocalDB.MSI**
 - If using a version of SQL Server other than LocalDB, each attendee must be able to use a SQL Server login (Windows or SQL Server authentication) with sufficient **permissions** to **create a new database** during the course
 - An **internet connection** is required to run any of the lab solution projects
 - Missing NuGet packages are restored on build to reduce lab file download size
-

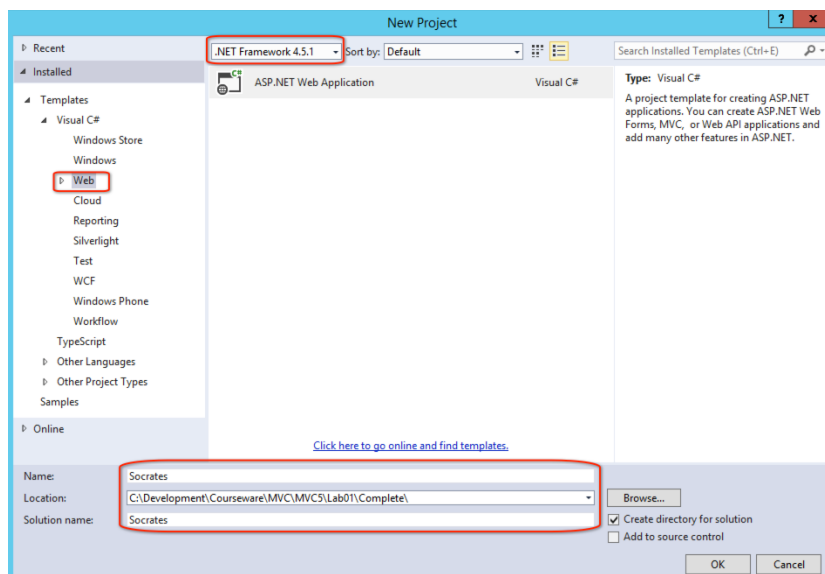
Lab I

Objectives

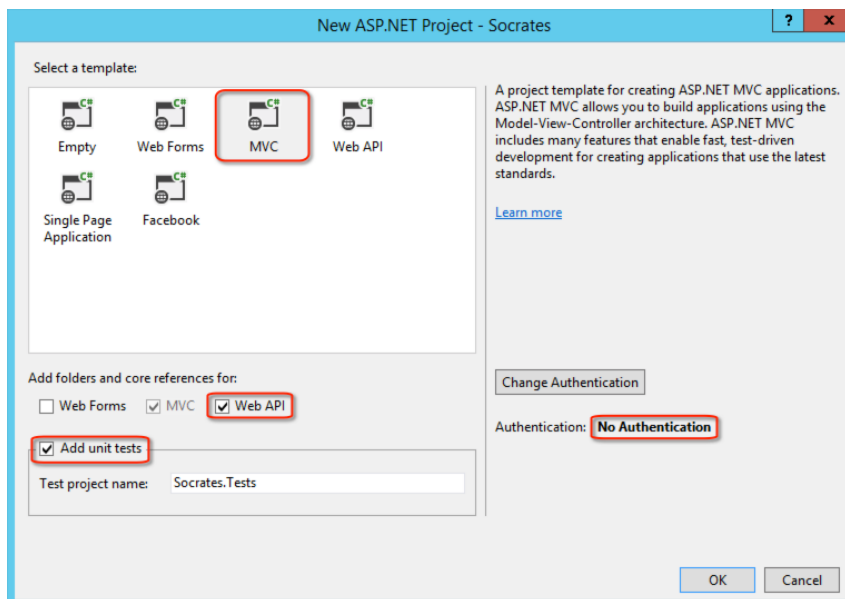
- Create a new ASP.NET MVC project
- Examine the folder structure and starter files


Procedures

- In Visual Studio, choose [**File > New > Project...**] from the menu.
 - Ensure **.NET Framework 4.5** (or later) is selected.
 - On the left side of the dialog, select [**Installed > Templates > Visual C# > Web**] and then choose the **ASP.NET Web Application** template.
 - Name the project **Socrates**.
 - Choose an appropriate location for the project. You will be working with this project throughout the course so make sure it is a location you can easily navigate to later.
 - Click **OK**.



2. In the New ASP.NET Application dialog that appears...
 - a. Select **MVC** as the template.
 - b. Check the box to add folders and core references for **Web API**.
 - c. Check the box to **add unit tests**.
 - d. Click the button labeled **Change Authentication** and choose **No Authentication**.
 - e. If there is a box labeled **Host in the cloud**, make sure that box is **not checked**.
 - f. Click **OK**.



 If the project was successfully created, you should be looking at the **Project_Readme.html** file included with the project that says “Your ASP.NET application” at the top.

3. Feel free to look at what’s in the Readme file but then close the tab, and **delete** the file



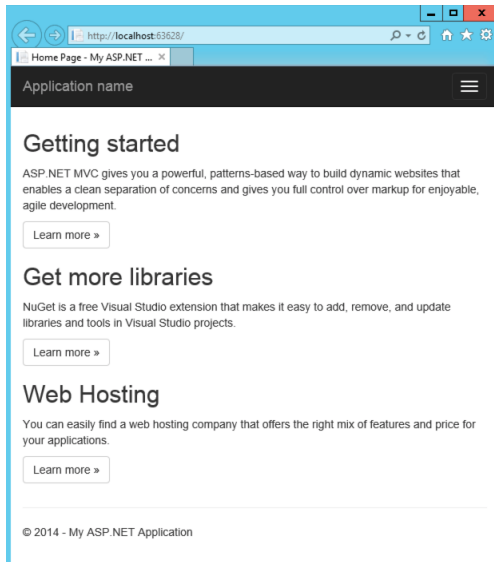
To delete a file, right-click on the file in **Solution Explorer** and select **delete**.


4. **Run** the application by choosing [**Debug > Start Debugging**] from the menu.



You could also use the start button in the toolbar or press F5.

5. With the homepage of the application open in the browser, **resize the window** and observe how the content is **reformatted** as the width of the window is made smaller.



 This reformatting behavior is provided for by the **Bootstrap** framework that is being used in the views created by Visual Studio.

6. Use the **menu** at the top of the page to navigate among the **home**, **about**, and **contact** pages.
7. Close the browser and return to Visual Studio.
8. Take some time to **examine** the folders and files that are in the project.



You have successfully completed Lab 1.


Lab 2

Objectives

- Add a simple action to `HomeController`
- Add a simple route
- Pass a parameter to an action
- Use a regular expression route constraint
- Use a custom route constraint


Procedures

1. If not already open, re-open your solution from Lab 1.

 If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 2.

2. Open **HomeController.cs** for editing. This file is located in the **Controllers** folder of the project.
3. Add a **new method** to `HomeController` that appears as follows:

```
public string Hello()
{
    return "Hello from HomeController";
}
```

 Notice that this method does not have a return type of `ActionResult` like the others. That's okay. We will talk about the `ActionResult` type a little later.

4. Open **RouteConfig.cs** for editing. This file is located in the **App_Start** folder.
5. **Comment-out** the default route that was created for you.

```
//routes.MapRoute(
//    name: "Default",
//    url: "{controller}/{action}/{id}",
//    defaults: new { controller = "Home", action = "Index", id = ... }
//);
```

6. Add a **new route** that for the `Hello` action that you just created.

```
routes.MapRoute(  
    name: "Hello",  
    url: "foo",  
    defaults: new { controller = "Home", action = "Hello" }  
);
```

7. **Run** the application. You should receive a **403** error page.

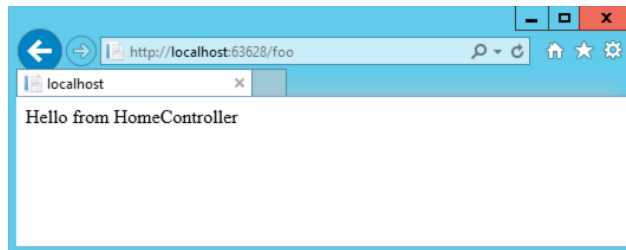


The reason you are receiving a 403 is because we have only specified one route and that route requires that the URL matches the pattern “foo”.

8. Change the URL to make a request for `/foo`. You should see a page with the string “Hello from HomeController” displayed.



The URL should look something like <http://localhost:63628/foo> but the port number (63628) chosen by IIS Express is probably different on your machine.



9. Close the browser and return to Visual Studio.
10. Open **HomeController.cs** for editing.
11. Modify the `Hello` action to take a parameter named **message** of type **string**. Also, change the code in the action to use that parameter.

```
public string Hello(string message)  
{  
    return message + " from HomeController";  
}
```

12. **Run** the application again and make a request for `/foo?message=Goodbye`



Notice that we were able to supply and receive a parameter from the request without modifying the routing configuration. We will see later that form field values can also be dealt with in a similar manner.

13. Close the browser and return to Visual Studio.
14. Open **RouteConfig.cs** for editing.
15. Modify the existing route to accept the message parameter as a **URL segment**.

```
routes.MapRoute(  
    name: "Hello",  
    url: "foo/{message}",  
    defaults: new { controller = "Home", action = "Hello" }  
);
```

16. **Run** the application again and make a request for **/foo/Greetings**
17. Modify the URL to make a request for **/foo**. You should receive a **404** error.



The reason for this is because the only available route requires a URL with **two components** and there is no default value for the message parameter.

18. Close the browser and return to Visual Studio.
19. Modify the existing route so that there is a **default parameter** for the **message** parameter.

```
routes.MapRoute(  
    name: "Hello",  
    url: "foo/{message}",  
    defaults: new { controller = "Home", action = "Hello", message = "Hi" }  
);
```

20. **Run** the application, make a request for **/foo**, and observe the results.



The next goal will be to add a constraint for this route using a regular expression. For this lab, we will specify that the message parameter must be at least two characters in length.

21. Close the browser and return to Visual Studio.
22. Add a **constraint** to the existing route using a **regular expression** that specifies that the **message** parameter must be **at least two characters in length**.

```
routes.MapRoute(  
    name: "Hello",  
    url: "foo/{message}",  
    defaults: new { controller = "Home", action = "Hello", message = "Hi" },  
    constraints: new { message = @".{2,}" }  
);
```


- 23.** Run the application and try a URL that satisfies the constraint and one that doesn't. You should receive a **404** response for URLs that do not satisfy the constraint (one character in length).
- 24.** Close the browser and return to Visual Studio.



The next objective is to implement a custom route constraint that uses C# code to evaluate the message parameter.

- 25.** Within **RouteConfig.cs**, define a new class named **MessageConstraint** that implements **IRouteConstraint**.

```
public class MessageConstraint : IRouteConstraint
{
    public bool Match(HttpContextBase httpContext, Route route, string parameterName,
        RouteValueDictionary values, RouteDirection routeDirection)
    {
        throw new NotImplementedException();
    }
}
```



To write the Match method, you can right-click on **IRouteConstraint** and choose **Implement Interface** (or use Ctrl-. with your cursor positioned somewhere in **IRouteConstraint**)



You could define this class in a different file or even in a different assembly.

- 26.** Replace the existing code in the Match method with code that applies the same rule we implemented with the regular expression (length >= 2).

```
public bool Match(HttpContextBase httpContext, Route route, string parameterName,
    RouteValueDictionary values, RouteDirection routeDirection)
{
    if (routeDirection == RouteDirection.IncomingRequest)
    {
        string message = values["message"] as string;
        return (message != null && message.Length >= 2);
    }
    return true;
}
```



We have to check if this is an incoming request because the routing system can also be used to generate links for use in views. We will see how to do this later.



Typically, you would only use an implementation of **IRouteConstraint** when you are unable to do the job with a regular expression.

- 27.** Modify the route to use the custom constraint instead of the regular expression.

```
routes.MapRoute(  
    name: "Hello",  
    url: "foo/{message}",  
    defaults: new { controller = "Home", action = "Hello", message = "Hi" },  
    constraints: new { message = new MessageConstraint() }  
);
```

- 28.** Run the application and check the results.



If you want to easily confirm that the constraint code is executing, you can set a **breakpoint** in the `Match` method before running the application.



You have successfully completed Lab 2.


Lab 3

Objectives

- Use `ContentResult` to return text from an action
- Return HTML from an action
- Return JSON from an action

Procedures

1. If not already open, re-open your solution from Lab 2.

 If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 3.

2. Open **HomeController.cs** for editing.
3. Modify the `Hello` action so that it has a return type of `ActionResult`.

```
public ActionResult Hello(string message)
```

4. Change the implementation of the `Hello` action so that a `ContentResult` object is created and returned.

```
public ActionResult Hello(string message)
{
    ContentResult cr = new ContentResult();
    cr.ContentType = "text/plain";
    cr.Content = String.Format("{0} from HomeController", message);
    return cr;
}
```

5. **Run** the application, make a request for **/foo/Hello** and examine the result.
6. Close the browser and return to Visual Studio.
7. Modify the `Hello` action to return some **HTML** content.

```
public ActionResult Hello(string message)
{
    ContentResult cr = new ContentResult();
    cr.ContentType = "text/html";
    cr.Content = String.Format("<h1>{0} from HomeController</h1>", message);
    return cr;
}
```



Specifying the **content type** is not strictly required here. The content type can normally be inferred automatically based on the content.

8. **Run** the application, make a request for **/foo/Hello** and examine the result.
9. Modify the `Hello` action to use the `Content` **convenience method**.

```
public ActionResult Hello(string message)
{
    string retVal = String.Format("<h1>{0} from HomeController</h1>", message);
    return Content(retVal);
}
```



Although this works, it is usually a better idea to define HTML content inside of a **View**. We will be doing this a little later in the course.

10. Close the browser and return to Visual Studio.
11. Modify the `Hello` action to return **JSON** content.

```
public ActionResult Hello(string message)
{
    var obj = new { FirstName = "John", LastName = "Doe", Message = message };
    return Json(obj);
}
```



We are using an instance of an **anonymous type** here but that is not a requirement. Any object can be serialized into JSON format using this technique.

12. **Run** the application and make a request for **/foo/Hello**. You should receive an **error**.



The error message says that the request was blocked because we are returning **JSON** data as part of a **GET** request. Returning JSON data via a GET request is disallowed by default to reduce the possibility exposing a **JSON hijacking vulnerability**.



For more information, see <http://haacked.com/archive/2009/06/25/json-hijacking.aspx/>



Since we are not returning a JSON array containing sensitive information, it is okay for us to allow this behavior.

13. Modify the `Hello` action so that returning JSON as part of a GET request is specifically allowed.

```
return Json(obj, JsonRequestBehavior.AllowGet);
```

14. Run the application, make a request for **/foo/Hello** and examine the result.



When receiving JSON data, IE will treat it like a file and prompt you to open or save the file.



For testing and debugging purposes, you may find it easier to use another browser (e.g. Google Chrome) or a network debugging proxy application like **Fiddler** to view JSON responses.



You have successfully completed Lab 3.

Lab 4

Objectives

- Add the necessary model objects to the project

Procedures

1. If not already open, re-open your solution from Lab 3.



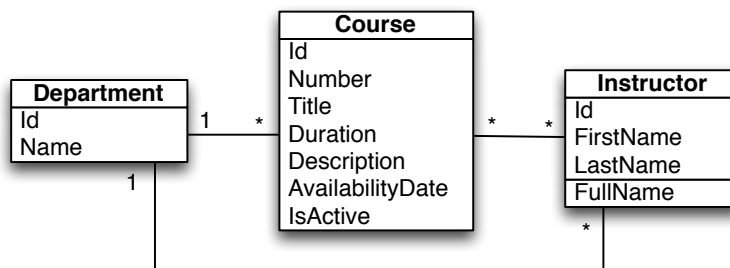
If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 4.

2. Right-click on the **Models** folder and choose [**Add > Existing Item...**]
3. Select the three files in [Lab Folder]\Code\Lab04 and click **Add**.



For **reusability** reasons, it is sometimes helpful to define the model objects in a **separate assembly**. Doing so here would be easy. We would just need to add a **reference** to the other assembly in our MVC project.

4. Take some time to **examine** the three classes you just added.



Notice that all of the **relationships** are represented in code with properties that are marked as **virtual**. This is necessary so that Entity Framework can override the implementation of each property and implement **lazy loading**. It is not necessary to mark these properties as **virtual** to use Entity Framework but we would not get automatic lazy loading.



You have successfully completed Lab 4.

Lab 5

Objectives

- Add a reference to Entity Framework 6
- Define a subclass of `DbContext`
- Use a custom connection string defined in `Web.config`
- Use a Data Annotation to identify required fields
- Use virtual model properties to enable lazy loading
- Write a unit test for `SocratesContext`

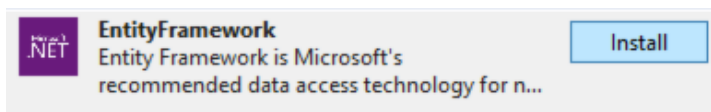
Procedures

1. If not already open, re-open your solution from Lab 4.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 5.

2. Right-click on the solution and choose [**Manage NuGet Packages for Solution...**]
3. Select **Entity Framework** in the search results and click **Install**. If not visible, select **All** from the **Online** section on the left and use the search box in the upper-right.



4. In the dialog that appears, ensure that **both projects** are checked and click **OK**.



These steps will cause the Entity Framework assembly to be downloaded to the solution's **packages** folder and a reference added to each project.

5. Add a new folder to the **Socrates** project named **DataAccess**.
6. Right-click on the **DataAccess** folder and choose [**Add > Existing Item...**]
7. Select the three files in [**Lab Folder**]\Code\Lab05 and click **Add**.



To create a `SocratesContext` object, we will need to provide a database connection string.



For easier maintenance, the application's config file is a good place to store the database connection string since it will allow us to change the connection string without having to recompile and redeploy our application.

8. Open **Web.config** for editing. Make sure you are opening the application's **root Web.config** file and not the one located in the Views folder.
9. Add a **connectionStrings** section to the Web.config file between the **configSections** element and the **appSettings** element. Leave the actual connection string blank for now.

```
...
</configSections>
<connectionStrings>
  <add name="ConnStr"
        providerName="System.Data.SqlClient"
        connectionString="" />
</connectionStrings>
<appSettings>
...
```

10. For the connection string itself, the value you should provide depends on your environment. For **LocalDB**, use the following...

```
Data Source=(localdb)\v11.0;Initial Catalog=Socrates;Integrated Security=True
```



Before we try to use our new `SocratesContext` class, we will add some data annotations to our model classes to identify the properties that should be required. This will cause EF to make sure those fields **cannot be null** in the database.

11. Open **Course.cs** for editing and use a **data annotation** to mark **Number**, **Title**, **Duration**, and **AvailabilityDate** as required. You will need to add a **using** directive for **System.ComponentModel.DataAnnotations**

```
[Required]
public string Number { get; set; }
[Required]
public string Title { get; set; }
[Required]
public float Duration { get; set; }
...
[Required]
public DateTime AvailabilityDate { get; set; }
```



Note that we didn't mark `IsActive` as required. However, because this field is not a **nullable boolean**, EF will mark this field as non-nullable in the database.

12. In **Department.cs**, mark the **Name** field as **required**.

13. In **Instructor.cs**, mark the **FirstName** and **LastName** fields as **required**.

14. **Build** the solution and ensure that you don't have any errors or warnings.



At this point, we would like to test the functionality of `SocratesContext`. We could start building controllers and views. However, instead, we will write a **unit test** for that task.

15. Right-click on the **Socrates.Tests** project, choose [**Add > New Folder**], and name the folder **DataAccess**.

16. Right-click on the **DataAccess** folder and choose [**Add > Unit Test...**]

17. Rename the new file **UnitTest1.cs** to **SocratesContextTests.cs** and click yes when asked to **rename all references**.

18. Rename the method `TestMethod1` to **GetAllDepartments**.

19. Add code to the `AllDepartments` method to **obtain the connection string** from the application's configuration file. You will need a **using** directive for **System.Configuration**

```
string connStr = ConfigurationManager.ConnectionStrings["ConnStr"].ConnectionString;
```



If this was the MVC application, this code would use **Web.config**. Here, this code will use the **App.config** file in the unit test project. So, we need to make sure the connection string is there.

20. Copy the **connectionStrings** section from the **Web.config** file and paste it into the **App.config** file of the **Socrates.Tests** project. If there is already an empty `connectionStrings` section, be sure to replace or remove it.

21. Implement the **AllDepartments** method so that it obtains an instance of `SocratesContext`, obtains the list of departments, and confirms that the collection is **not null**. You will need **using** directives for **Socrates.DataAccess** and **System.Linq**

```
// Arrange
string connStr = ConfigurationManager.ConnectionStrings["ConnStr"].ConnectionString;
var context = SocratesContextFactory.GetContext(connStr);

// Act
var departments = context.GetAllDepartments().ToList();

// Assert
Assert.IsNotNull(departments);
```



The call to **ToList** here ensures that the objects are fetched from the database immediately and the operation is not **deferred**.

22. From the Visual Studio menu, choose [**Test > Windows > Test Explorer**]

23. At the top of the **Test Explorer** window, click the link for **Run All** and wait for the test to complete. Since the database does not exist yet, it may take a few moments for EF to create the database.



If the test completed successfully, the database for our application should now exist. So, we should take a look at what was created.

24. From the Visual Studio menu, choose [**View > SQL Server Object Explorer**]



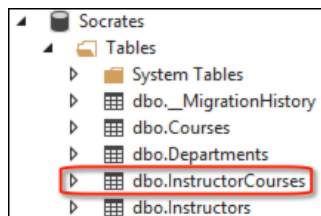
If **SQL Server Object Explorer** does not immediately appear, it may already be open and available as a **tab** in the IDE.

25. In **SQL Server Object Explorer**, expand the SQL Server node and check to see if an item already exists for the server you used in your **connection string**. If not, right-click on **SQL Server**, choose **Add SQL Server...**, and connect to your server. If using **LocalDB**, you should use **(localdb)\v11.0** for the server name.

26. Examine the tables and columns of the **Socrates** database. Check the **required fields** and look for both **primary** and **foreign keys**.



Notice that EF created a table named **InstructorCourses** to provide for the **many-to-many** relationship between instructors and courses.



You have successfully completed Lab 5.

Lab 6

Objectives

- Import some sample data

Procedures

1. From the Visual Studio menu, select [**File > Open > File...**] and select the **SampleData.sql** file from [**Lab Folder**]\Code
2. Click the **Connect** toolbar button in the SQL editor window and provide the correct server name. If using LocalDB, use **(localdb)\v11.0**
3. Select **Socrates** from the list of available databases.
4. Click the **Execute** toolbar button to execute the SQL.
5. Use **SQL Server Object Explorer** to verify that data is now present. To do this, right-click on a table and select [**View Data**]



You have successfully completed Lab 6.

Lab 7

Objectives

- Modify the homepage for the application

Procedures

1. If not already open, re-open your solution from Lab 5.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 7.



Before working on the application's homepage, we first need to restore the default route we disabled in a previous lab.

2. Open **RouteConfig.cs** for editing.
3. Un-comment the **default route**. It is okay to leave the additional route for **foo/{message}**

```
routes.MapRoute(  
    name: "Default",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
);
```



Notice that if no **controller** or **action** is provided, the default values indicate that the **Index** action of **HomeController** should be invoked.

4. Open **HomeController.cs** for editing and examine the **Index** action.

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



The code for this action calls the **View** convenience method and doesn't specify a view name. Therefore, it will look in the **Views** folder for a folder named **Home** and then look for a view named **Index**.

5. Open **Views/Home/Index.cshtml** for editing.



An easy way to open the view for an action is to **right-click** in the code for an action and select **Go To View**. You can also **right-click** in a view and select **Go To Controller**.

6. Modify the `<div>` element that is using the Bootstrap **jumbotron** class. Leave the **Learn more** link in place but replace the URL with `#` to act as a placeholder.

```
<div class="jumbotron">
  <h1>Acme Training</h1>
  <p class="lead">Acme is a leading technology training company</p>
  <p><a href="#" class="btn btn-primary btn-large">Learn more &raquo;</a></p>
</div>
```

7. Modify the text of the three **columns** to provide information about **departments**, **courses**, and **instructors**.

```
<div class="col-md-4">
  <h2>Departments</h2>
  <p>We offer training across a wide variety of technologies including Microsoft,
    Apple, Java, Oracle, JavaScript, and more.</p>
  <p><a class="btn btn-default" href="#">Learn more &raquo;</a></p>
</div>
<div class="col-md-4">
  <h2>Courses</h2>
  <p>All of our courses include up-to-date technical information and hands-on lab
    exercises.</p>
  <p><a class="btn btn-default" href="#">Learn more &raquo;</a></p>
</div>
<div class="col-md-4">
  <h2>Instructors</h2>
  <p>All of our instructors have real-world experience and extensive teaching
    experience.</p>
  <p><a class="btn btn-default" href="#">Learn more &raquo;</a></p>
</div>
```

8. **Run** the application and **examine** the appearance of the modified homepage. Be sure to change the **width** of the browser and confirm that the **adaptive** capabilities provided by Bootstrap are still working correctly.



Notice that the title in the upper-left still says **Application name** and the footer says **My ASP.NET Application**. These elements are not defined in `Index.cshtml`. Since they are site-wide elements, they are defined in `_Layout.cshtml`

9. Return to Visual Studio and open `_Layout.cshtml` for editing.
10. Change the **title** element to use the company name.

```
<title>@ViewBag.Title - Acme</title>
```

11. Change the use of **Application name** to the company name.

```
@Html.ActionLink("Acme Training", "Index", "Home", null, new { @class = ...
```



We have not covered the use of `Html.ActionLink` yet. We are just changing the text right now. We will talk more about the other parameters a little later.

12. Change the content of the **footer** element to use the company name.

```
<footer>  
<p>&copy; @DateTime.Now.Year - Acme, Inc.</p>  
</footer>
```

13. **Run** the application and examine the results.



You have successfully completed Lab 7.

Lab 8

Objectives

- Modify the homepage view so links are generated using **Url.Action**

Procedures

1. If not already open, re-open your solution from Lab 7.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 8.

2. Open **Views/Home/Index.cshtml** for editing.
3. Modify the link in the **jumbotron** element so that **Url.Action** is used to generate the URL.

```
<a href="@Url.Action("About")" class="btn btn-primary btn-large">Learn more ...
```



Notice that only the **action** name is required here. This is because the **About** action exists in the **same controller** that is returning this view.

4. **Run** the application and **examine** the source for the generated link.

```
<a href="/Home/About" class="btn btn-primary btn-large">Learn more &raquo;</a>
```

5. Return to Visual Studio and modify the link in the **departments** section to use **Url.Action**

```
<a class="btn btn-default" href="@Url.Action("Index", "Department")">Learn more ...
```

6. **Run** the application and **examine** the source for the generated link.

```
<a class="btn btn-default" href="/Department">Learn more &raquo;</a>
```



Notice that ASP.NET MVC was able to generate a URL for the link **based on the default route** definition even though this controller and action **do not exist**.

7. Click the **Learn more** button in the **departments** section and confirm that you receive a **404 error**.

8. Return to Visual Studio and modify the link in the **courses** section and **instructors** section.

```
<a class="btn btn-default" href="@Url.Action("Index", "Course")">Learn more ...
```

...

```
<a class="btn btn-default" href="@Url.Action("Index", "Instructor")">Learn more ...
```



Of course, these controllers don't exist yet but these links will be ready when they do.



You have successfully completed Lab 8.

Lab 9

Objectives

- Create `DepartmentController`
- Create an **Index** action
- Define the Index view as a **strongly-typed view**
- Pass the collection of departments to the view
- Display the collection of departments as a **bulleted list**

Procedures

1. If not already open, re-open your solution from Lab 8.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 9.

2. Right-click on the **Controllers** folder and select [**Add > Controller...**]
3. In the dialog that appears, select **MVC 5 Controller - Empty**, click **Add**, and name the controller **DepartmentController**.



There are other options in the new controller dialog allow you to generate a more full-featured starting point. **MVC 5 Controller with read/write actions** will provide method stubs for all of the typical CRUD operations. **MVC 5 Controller with views, using Entity Framework** will attempt to create a fully functional controller with data access code and associated views based on a selected data context. For this lab, we will create the actions manually.

4. Open **DepartmentController.cs** for editing.



There is already an Index action that simply returns a view. We will modify this action to obtain the department objects from the database and pass them to the view.

5. Add the following code to the **Index** action. You will need a **using** directive for **System.Configuration** and **Socrates.DataAccess**

```
// GET: /Department/
public ActionResult Index()
{
    string connStr = ConfigurationManager.ConnectionStrings["ConnStr"].ConnectionString;
    var context = SocratesContextFactory.GetContext(connStr);
    var departments = context.GetAllDepartments().OrderBy(d => d.Name).ToList();

    return View(departments);
}
```



Notice the **comment** inserted by Visual Studio that specifies the **URL** that can be used to invoke this action. Keep in mind that this comment might become **incorrect** if you modify the application's **routing configuration**.

6. To add a view, **right-click** somewhere in the code for the **Index** action and select [**Add View...**]
7. In the dialog that appears, name the view **Index**, select **Empty (without model)** for the template, and click **Add**.



Once again, we could have decided to select a different template for a more full-featured starting point. We will do that for other view that we create later in the course.



Since we will be created several strongly-typed views, we will add a **namespace** to the **Web.config** file in the **Views** folder.

8. Open **Views\Web.config** for editing. Make sure you are not editing the application's root Web.config file.
9. Add a **namespace** element for **Socrates.Models**

```
<add namespace="Socrates.Models" />
```

10. Open **Views\Department\Index.cshtml** for editing and add a line to the top of the file that specifies that the model type for the view is **IEnumerable<Department>**

```
@model IEnumerable<Department>
```



If Intellisense does not recognize the namespace you added to the Web.config, close and re-open the view file.

- 11.** Add some HTML to **Index.cshtml** to display the **list of departments**.

```
@model IEnumerable<Department>
@{
    ViewBag.Title = "Departments";
}

<h2>Departments</h2>
<ul>
@foreach (Department dept in Model)
{
    <li>@dept.Name</li>
}
</ul>
```

- 12. Run** the application and check the results.



You have successfully completed Lab 9.


Lab 10


Objectives

- Move common controller code into a **base class**
- Override the controller's **Initialize** method

Procedures

1. If not already open, re-open your solution from Lab 9.

 If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 10.

 We currently have some code in `DepartmentController`'s **Index** action that we will need in many of our other controllers and actions (specifically, the code that initializes the `SocratesContext` object). We will avoid code duplication by using a controller base class.

2. Right-click on the **Controllers** folder, select [**Add > Controller...**], select **MVC 5 Controller - Empty**, and name the controller **SocratesController**.
3. Remove the **Index** action and add a **protected instance variable** named **context** of type **ISocratesContext**. You will need a **using** directive for **Socrates.DataAccess**

```
public class SocratesController : Controller
{
    protected ISocratesContext context;
}
```

4. Add a method that **overrides** the **Initialize** method and sets the context variable. You will need a **using** directive for **System.Configuration**

```
protected override void Initialize(System.Web.Routing.RequestContext requestContext)
{
    base.Initialize(requestContext);
    string connStr = ConfigurationManager.ConnectionStrings["ConnStr"].ConnectionString;
    context = SocratesContextFactory.GetContext(connStr);
}
```

5. Open **DepartmentController.cs** for editing.
6. Change the definition of `DepartmentController` so that it inherits from **SocratesController**.

```
public class DepartmentController : SocratesController
```

7. Modify the **Index** action so that it uses the base class instance variable. In this case, you can simply remove the first two lines of code.

```
public ActionResult Index()
{
    var departments = context.Departments.OrderBy(d => d.Name).ToList();
    return View(departments);
}
```

8. **Run** the application and confirm that the department listing page works as before.



You have successfully completed Lab 10.

Lab 11

Objectives

- Create an action and view for editing a department

Procedures

1. If not already open, re-open your solution from Lab 10.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 11.

2. Open **DepartmentController.cs** for editing.
3. Add a new action named **Edit** that will allow for the user to **edit an existing department**. You will need to add a **using** directive for **Socrates.Models**

```
public ActionResult Edit(int id)
{
    Department dept = context.GetAllDepartments().SingleOrDefault(d => d.Id == id);
    if (dept == null) return HttpNotFound();
    return View(dept);
}
```



Notice that the method takes an integer parameter named **id**. The **default route** already accepts this value as part of the URL.

4. Right-click somewhere in the code for Edit and select [**Add View...**]
5. In the dialog that appears, name the view **Edit**, select a template of **Edit**, select a model class of **Department**, leave the **Data context class** blank, check the box for **Reference script libraries**, and click **Add**.

6. Spend a moment to **examine** the code in **Views\Department\Edit.cshtml**



We could run the application and check the appearance of the form by manually typing in the correct URL. However, instead, we will modify the department list page so each department appears as a **link** to the edit page for that department.

7. Open **Views\Department\Index.cshtml** for editing.
8. Replace the code that outputs each department name with an appropriate use of **Html.ActionLink**

```
@foreach (Department dept in Model)
{
    <li>@Html.ActionLink(dept.Name, "Edit", new { id = dept.Id })</li>
}
```



Notice the **third parameter** used in the call to **Html.ActionLink**. This parameter allows you to specify **route values**. The value may appear as part of the **URL** or as a **query string** value depending on the application's routing configuration.

9. **Run** the application.
10. Go to the **department list page** and then **click one of the links**. You should see a **form** with the department name in a text field and a **Save** button.
11. **Change** the name of the department and click **Save**. It should look like the page simply reloaded without the change being saved.



Submitting the form right now just makes another request for the **Edit** action which returns the form again. We need to write code to determine what should happen when the form is submitted. This will be done in the **next lab**.



You have successfully completed Lab 11.

Lab 12

Objectives

- Create an **action** for the submission of the department edit form

Procedures

1. If not already open, re-open your solution from Lab 11.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 12.

2. Open **DepartmentController.cs** for editing.
3. Add a second Edit action. This one should use an HttpPost action filter and accept a FormCollection parameter.

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
}
```



Notice that the first parameter is the **id** for the department being edited. This is just for convenience. This value is also present in the form.

4. Implement the **Edit** action so that it uses the **context** object to save the changes.

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
    var dept = new Department();
    if (TryUpdateModel(dept, collection)) {
        context.MarkAsModified(dept);
        context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View();
}
```



We are simply returning the View again when validation fails. This will work in this case because we are using **HTML helpers** to generate the form fields and the helpers know to use the **ModelState** property to populate the fields if the model is **invalid**.



Also note that we are using the **context** to set the state of the object to **modified**. This allows us to update the object without making **two trips** to the database.

5. **Run** the application and **modify the name of a department**. When redirected back to the list, the name of the department should reflect the new value.



You have successfully completed Lab 12.

Lab 13

Objectives

- Add the ability to create a new department

Procedures

1. If not already open, re-open your solution from Lab 12.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 13.



The first step is to **add a link** to the department list page that will allow the user to navigate to a page where they can create a new department.

2. Open **Views\Department\Index.cshtml** for editing and add a link above the list of departments.


```
<h2>Departments</h2>
<p>@Html.ActionLink("Create New", "Create")</p>
<ul>
...
```

3. Open **DepartmentController.cs** for editing and add the code for both **Create** actions (**GET** and **POST**).


```
public ActionResult Create()
{
    return View(new Department());
}


[HttpPost]
public ActionResult Create(FormCollection collection)
{
    var dept = new Department();
    if (TryUpdateModel(dept, collection))
    {
        context.MarkAsAdded(dept);
        context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View();
}
```

4. **Add a view** for the **Create** action using the **Create template**.
5. **Run** the application and create a new department.
6. Try to create another department and **leave the name blank** this time. You should receive a **validation error** message stating that the field is **required**.



The screenshot shows a web form with a text input field. To the right of the input field, a red error message reads "The Name field is required." Below the input field is a button labeled "Create".

 What is happening to make this work will be discussed shortly.

 You have successfully completed Lab 13.

Lab 14

Objectives

- Use a **partial view** to eliminate duplicate view code

Procedures

1. If not already open, re-open your solution from Lab 13.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 14.



We currently have both an **edit view** and a **create view** for departments. If you examine those two views, you should notice that there is a section of code that is **identical** in both views. It is possible to eliminate that duplication by moving that code into a **partial view**.

2. Right-click on the **Views\Department** folder, select [**Add > MVC 5 Partial Page (Razor)**], and use a name of **_DepartmentForm**
3. Specify the **model** for the partial view as **Department**.

@model **Department**

4. **Copy** the common view code from **Views\Department\Edit** and **paste** it into the new partial view. The common code in this case is the `<div>` element with the CSS class of **form-group**.

```
<div class="form-group">
  @Html.LabelFor(model => model.Name, new { @class = "control-label col-md-2" })
  <div class="col-md-10">
    @Html.EditorFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)
  </div>
</div>
```

5. Replace the `<div>` in **Views\Department\Edit** and **Views\Department\Create** with a call to **Html.Partial**

@Html.Partial("_DepartmentFormPartial")

6. **Run** the application and **confirm** that both forms still look as they did before.



You have successfully completed Lab 14.

Lab 15

Objectives

- Create `CourseController`
- Create an **action** and a **view** to display the **list of courses**
- Implement the ability to view the **details** for a course

Procedures

1. If not already open, re-open your solution from Lab 14.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 15.

2. Add a new controller to the application named **CourseController** that inherits from `SocratesController`



For the course listing page, the requirement is to list the courses in **groups** according to their department.

3. Implement the **Index** action so that it retrieves a list of the **departments**.

```
public ActionResult Index()
{
    var departments = context.GetAllDepartments().OrderBy(d => d.Name).ToList();
    return View(departments);
}
```

4. Use what you have learned so far and create a **view** for the list of courses that appears as shown on the next page. The course title should be configured as a **link** that allows the user to view the **details** for that course.



If you would like to display a pencil icon, you can use a glyph that is available as part of the Bootstrap framework.

```
<span class="glyphicon glyphicon-pencil"></span>
```

Courses	
Add New Course	
.NET	
MVC-104:	Introduction to ASP.NET MVC 5 [4 days]
NET-944:	Introduction to Visual C++/CLI [5 days]
NET-942:	.NET Web Application Security [3 days]
MVC-102:	Introduction to ASP.NET MVC 4 [4 days]
Apple	
MBL-120:	Building iOS 7 Applications [5 days]
Java	
JWS-202:	Developing Java Web Services for Java EE 6 [5 days]

5. Add an action named **Detail** to **CourseController**

```
public ActionResult Details(int id)
{
    var course = context.GetAllCourses().SingleOrDefault(c => c.Id == id);
    if (course == null) return HttpNotFound();
    return View(course);
}
```

6. Add a view for the **Detail** action. Use the **Detail template**. Modify the view so the end result appears as follows:

Details	
Course	
<hr/>	
Department	JavaScript
Number	SCRPT-136
Title	Introduction to AngularJS
Duration	3 days
Description	Teaches developers how to use AngularJS to facilitate development of single-page web applications.
Available	9/27/2013
Is Active	<input checked="" type="checkbox"/>
Edit Back to List	



For the **label** and **format** of **availability date**, you can modify the view or add **Data Annotations** to the model.

```
[Display(Name="Available")]
[DisplayFormat(DataFormatString="{0:d}", ApplyFormatInEditMode=true)]
public DateTime AvailabilityDate { get; set; }
```



You have successfully completed Lab 15.

Lab 16

Objectives

- Implement the ability to **edit a course**
- Use a **ViewModel** to provide the **course** and the **list of departments** to the view

Procedures

1. If not already open, re-open your solution from Lab 15.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 16.

2. Add a **new folder** to the project named **ViewModels**
3. Add a **new class** to the **ViewModels** folder named **CourseEditViewModel**
4. Add properties to **CourseEditViewModel** for the **course**, the **list of departments**. You will need to add a **using** directive for **Socrates.Models**.

```
public class EditCourseViewModel
{
    public Course Course { get; set; }
    public IEnumerable<Department> Departments { get; set; }
}
```



The view will require a collection of **SelectListItem** objects. So, let's write a read-only property in the ViewModel to provide that.



Remember, **adapting** model data for consumption by the view is a typical responsibility of a **ViewModel**.

5. Add a **read-only property** named **DepartmentList** to **EditCourseViewModel** to provide what is required. Make sure to add a **using** directive for **System.Web.Mvc** and NOT **System.Web.WebPages.Html**.


```
public IEnumerable<SelectListItem> DepartmentList
{
    get
    {
        return from d in Departments
               select new SelectListItem()
               {
                   Value = d.Id.ToString(),
                   Text = d.Name,
                   Selected = (d.Id == Course.Department.Id)
               };
    }
}
```

6. Add an **Edit** action to **CourseController** that populates an instance of **CourseEditViewModel** and sends it to the view. You will need to add a **using** directive for **Socrates.ViewModels**.

```
public ActionResult Edit(int id)
{
    var ecvm = new CourseEditViewModel();
    ecvm.Course = context.GetAllCourses().SingleOrDefault(c => c.Id == id);
    if (ecvm.Course == null) return HttpNotFound();

    ecvm.Departments = context.GetAllDepartments().OrderBy(d => d.Name).ToList();
    return View(ecvm);
}
```


7. Add a **view** for the **Edit** action. Select the **Edit template** and make sure to select **Course** (not **CourseEditViewModel**) as the model type.

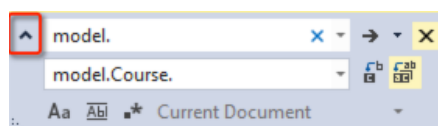
 We will be changing the model type to **CourseEditViewModel** but selecting **Course** helps to generate a better **starting point** for the view.

8. Change the **model** of the **Edit** view to be **CourseEditViewModel**.

```
@model Socrates.ViewModels.EditCourseViewModel
```

9. Replace all occurrences of **model.** with **model.Course.**

 You can use **ctrl-f** to bring up the find box and click the caret to expand the replace option.



- 10.** Add a form-group for a **department** drop-down before any other form-group.

```
<div class="form-group">
  <label class="control-label col-md-2" for="DepartmentId">Department</label>
  <div class="col-md-10">
    @Html.DropDownList("DepartmentId", Model.DepartmentList)
  </div>
</div>
```

- 11.** Run the application, open the **details** for a course, and click the **Edit** link.

- 12.** Confirm that the **correct department** is selected in the drop-down.

- 13.** View the **source** for the page and notice how the **form fields** are named.

```
<input id="Course_Number" name="Course.Number" ... />
...
<input id="Course_Title" name="Course.Title" ... />
```

- 14.** Close the browser and return to Visual Studio.

- 15.** Add the **POST** version of the **Edit** action.

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection, int departmentId)
```

- 16.** Implement the code to **save the changes** for the course.

```
var course = context.GetAllCourses().SingleOrDefault(c => c.Id == id);
if (TryUpdateModel(course, "Course", collection)) {
    course.Department = context.GetAllDepartments().SingleOrDefault(d => d.Id == departmentId);
    context.SaveChanges();
    return RedirectToAction("Details", new { id = id });
}
// validation failed - repopulate ecvm
var ecvm = new CourseEditViewModel();
ecvm.Course = course;
ecvm.Departments = context.GetAllDepartments().OrderBy(d => d.Name).ToList();
return View(ecvm);
```



Notice how we are fetching the course using the context so that we can modify the department that it's related to. There is a little overhead involved with this but EF is managing all of the foreign keys for us.

- 17.** Run the application and test the functionality of the course edit form. Try changing a **duration** and then try changing the **department** for a course.



You have successfully completed Lab 16.


Lab 17


Objectives

- Add **data validation** for editing a course
- Use a **validation attribute**
- Implement custom **server-side validation**

Procedures

1. If not already open, re-open your solution from Lab 16.


 If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 17.


 The first validation rule we will implement is that all course numbers must be a series of letters followed by a dash followed by a series of numbers.

2. Open **Course.cs** for editing.
3. Add a **regular expression** validation attribute to the **Course** property for the rule described at the beginning of the lab.

```
[Required]  
[RegularExpression(@"[A-Z]+-\d+")]  
public string Number { get; set; }
```

4. **Run** the application and try to set the number of a course to an **invalid** value. Also try to omit the number altogether and notice the **required** validation still works.

 The field Number must match the regular expression '[A-Z]+-\d+'.

 Obviously, this is not the most friendly error message. Let's change it.

5. Return to Visual Studio and add an **ErrorMessage** parameter to the RegularExpression validation attribute.

```
[RegularExpression(@"[A-Z]+-\d+", ErrorMessage="Must be in the format AAA-123")]
```

6. **Run** the application and check the results.



The next validation rule we want to implement is that you should not be able to use a course number that is already assigned to another course. This is not possible to implement with a regular expression. We'll implement this logic in the controller action.

7. Open **CourseController.cs** for editing.
8. Modify the **POST** version of the **Edit** action to check if the course number is **already in use**. The code below shows what the modified code should look like.

```
var course = context.GetAllCourses().SingleOrDefault(c => c.Id == id);

bool isValid = TryUpdateModel(course, "Course", collection);
if (isValid && (context.GetAllCourses().Count(c => c.Number == course.Number && c.Id != course.Id) > 0))
{
    isValid = false;
    ModelState.AddModelError("Course.Number", "Course number already in use");
}
if (isValid) {
    course.Department = context.GetAllDepartments().SingleOrDefault(d => d.Id == departmentId);
    context.SaveChanges();
    return RedirectToAction("Details", new { id = id });
}
// validation failed - repopulate ecvm
var ecvm = new CourseEditViewModel();
ecvm.Course = course;
ecvm.Departments = context.GetAllDepartments().OrderBy(d => d.Name).ToList();
return View(ecvm);
```

9. **Run** the application and try to assign a course number to a course that is **already assigned** to another course. You should see the validation error when trying to submit the form.



You have successfully completed Lab 17.

Lab 18

Objectives

- Add the ability to **search** for courses on the homepage
- Use **Ajax** to perform the search and display the results **without reloading the page**

Procedures

1. If not already open, re-open your solution from Lab 17.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 18.

2. Use **NuGet** to install the package named **Microsoft jQuery Unobtrusive Ajax**. You only need to add it to the **Socrates** project and not **Socrates.Tests**.
3. Open **Views\Shared_Layout.cshtml** and insert an additional call to **Scripts.Render** after the one for Bootstrap.

```
@Scripts.Render("~/Scripts/jquery.unobtrusive-ajax.js")
```

4. Open **Views\Home\Index.cshtml** for editing and add the following code to the bottom of the view (after the last closing div).

```
<br />
<div class="row">
  <div class="col-md-12">
    @using (Ajax.BeginForm("Search", "Course", null, new AjaxOptions {
      HttpMethod = "GET",
      InsertionMode = InsertionMode.Replace,
      UpdateTargetId = "searchResults" },
      new { @class = "form-inline" })))
    {
      <input class="form-control" name="q" />
      <button type="submit" class="btn btn-default">Search</button>
    }
  </div>
</div>
<br />
<div class="row">
  <div class="col-md-12" id="searchResults">
  </div>
</div>
```



Notice in the call to `Ajax.BeginForm` that we will be calling an action named **Search** in

CourseController. Also notice that we will be replacing the contents of the **searchResults** `<div>` with whatever comes back from the server.

5. Open the code for **CourseController** and add an action named **Search** that takes a **string** parameter named **q**.

```
public ActionResult Search(string q)
{
}
```



The requirement for this method is to return a chunk of **HTML** containing **search results** that can be inserted into the page.

6. Add code to the **Search** action that obtains the course objects whose **Title** contains the string provided.

```
var courses = context.GetAllCourses().Where(c => c.Title.Contains(q)).ToList();
```

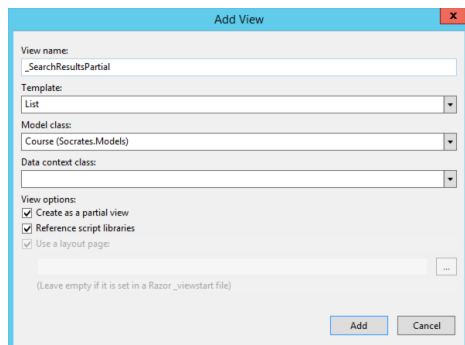
7. Add code that will return an **appropriate message** if there are **no matching courses**.

```
if (courses.Count() == 0)
{
    return Content("<strong>No matching courses found</strong>");
}
```

8. Finally, if there are **matching courses**, send the list of courses to a **partial view**. Here is the code for the entire action:

```
public ActionResult Search(string q)
{
    var courses = context.GetAllCourses().Where(c => c.Title.Contains(q)).ToList();
    if (courses.Count() == 0) {
        return Content("<strong>No matching courses found</strong>");
    }
    return PartialView("_SearchResults", courses);
}
```

9. Add a **partial view** to the **Views/Course** folder named **_SearchResults**. Use the **List template** and a model class of **Course**.



10. In the view, **delete** the link for creating a new course and **delete** all of the table columns except the ones for **number**, **title**, and **duration**.
11. **Run** the application and try a few queries.
12. As a bonus exercise, make the course **titles** display as **links** that take you to the **details** for that course.

net

Search

Number	Title	Duration
MVC-104	Introduction to ASP.NET MVC 5	4
NET-942	.NET Web Application Security	3
MVC-102	Introduction to ASP.NET MVC 4	4



You have successfully completed Lab 18.

Lab 19

Objectives

- Add a jQuery UI **pop-up calendar** for **course availability date**

Procedures

1. If not already open, re-open your solution from Lab 18.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 19.

2. Use **NuGet** to install the package named **jQuery UI (Combined Library)**
3. Open **BundleConfig.css** in the **App_Start** folder and add the jQuery UI stylesheet to the **css** bundle (listed last). The modified statement is shown below:

```
bundles.Add(new StyleBundle("~/Content/css").Include(  
    "~/Content/bootstrap.css",  
    "~/Content/themes/base/all.css",  
    "~/Content/site.css"));
```



Bundles provide a way to combine multiple files into one downloadable resource. The **css** bundle is included in the **_Layout** view.



We are adding all of the jQuery UI CSS here. In a real-world application, it would be better to only include the CSS for the parts of jQuery UI that you are actually using.

4. Add a **new bundle** for the jQuery UI JavaScript file.

```
bundles.Add(new ScriptBundle("~/bundles/jqueryui").Include(  
    "~/Scripts/jquery-ui-{version}.js"));
```

5. Open **_Layout.cshtml** for editing.
6. Near the bottom, after the last call to **Scripts.Render**, add a new line for the jQuery UI bundle and for another JavaScript file that we will be creating.

```
@Scripts.Render("~/bundles/jqueryui")  
@Scripts.Render("~/Scripts/socrates.js")
```

7. Right-click on the **Scripts** folder, choose [**Add > JavaScript File**], and name the file **socrates**.



Instead of attaching a datepicker widget to one specific input element, we will use jQuery to attach the widget to **any input element** with a **type attribute** of **datetime**.

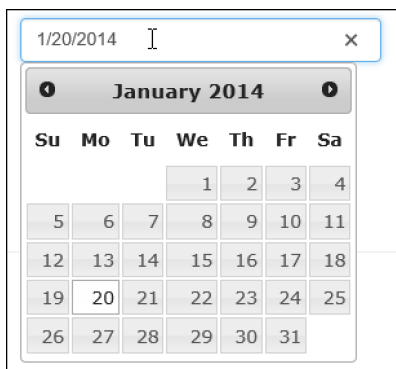


When an **HTML helper** renders an element for a **DateTime** field, it automatically adds the **type attribute** to the element and sets its value to **datetime**.

8. Add the following code to **socrates.js**

```
$(document).ready(function () {  
    $(":input[type='datetime']").each(function () {  
        $(this).datepicker();  
    });  
})
```

9. **Run** the application, choose to **edit** a course, and click in the **availability date** field to see the results.



If the calendar appears but does not display correctly, you are missing the CSS. Make sure you are including it correctly in the bundle.



You have successfully completed Lab 19.

Lab 20

Objectives

- Use a jQuery UI **autocomplete widget** to add autocomplete functionality to the **search form**

Procedures

1. If not already open, re-open your solution from Lab 19.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 20.



Like we did with the datepicker widget, we will use **unobtrusive** JavaScript to attach the autocomplete widget to any element that has a specific attribute value.

2. Add the following statement to the document ready function in **socrates.js**.

```
$( ":input[data-autocomplete]" ).each( function () {  
    $( this ).autocomplete( { source: $( this ).attr( "data-autocomplete" ) } );  
});
```



Notice that this code will attach an autocomplete widget to an element that has an attribute named **data-autocomplete**. It will use the value of the attribute as the **source** parameter for the widget.

3. Open **Views\Home\Index.cshtml** for editing and add the **data-autocomplete** attribute to the search input element.

```
<input class="form-control" name="q"  
    data-autocomplete="@Url.Action("QuickSearch", "Course")" />
```

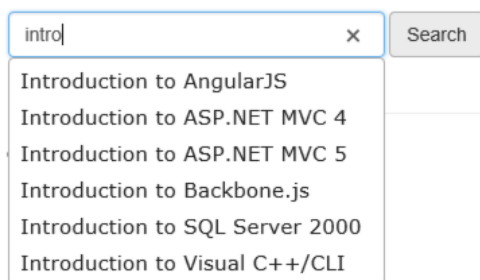



The last thing we will need to do is implement the **QuickSearch** action in **CourseController**.


4. Add an action to **CourseController** named **QuickSearch** that takes a **string** parameter named **term** and returns a list of matching **titles** in **JSON** format. The code is shown on the following page.

```
public ActionResult QuickSearch(string term)
{
    var courseTitles = new List<string>();
    if (!String.IsNullOrEmpty(term))
    {
        courseTitles = (from c in context.GetAllCourses()
                        where c.Title.Contains(term)
                        orderby c.Title
                        select c.Title).ToList();
    }
    return Json(courseTitles, JsonRequestBehavior.AllowGet);
}
```

5. Run the application and check the functionality.



 If the drop-down does not appear, the browser may be using a **cached** version of **socrates.js**. Try reloading the page in the browser to fix the issue.

 You have successfully completed Lab 20.

Lab 21

Objectives

- Create a **custom filter**
- Apply custom filter to a **controller**
- Register a custom filter as a **global filter**

Procedures

1. If not already open, re-open your solution from Lab 20.



If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 21.

2. Add a **new folder** to the project named **Filters**
3. Add a new class to the **Filters** folder named **TimestampAttribute** that inherits from `ActionFilterAttribute`. You will need a **using** directive for **System.Web.Mvc** (not **System.Web.Http.Filters**).

```
public class TimestampAttribute : ActionFilterAttribute
{
}
```

4. Override **OnActionExecuting** and **OnActionExecuted** so that the **current time** is written to the response stream.

```
public override void OnActionExecuting(ActionExecutingContext filterContext)
{
    filterContext.HttpContext.Response.Write(DateTime.Now.ToString("HH:mm:ss.ffff") + " - ");
}

public override void OnActionExecuted(ActionExecutedContext filterContext)
{
    filterContext.HttpContext.Response.Write(DateTime.Now.ToString("HH:mm:ss.ffff"));
}
```




Writing directly to the response within a filter is typically not a good idea in a production application. Instead, you should consider writing performance and audit information to a file.

5. Add the **Timestamp** attribute to **CourseController**. You will need a **using** directive for **Socrates.Filters**

```
[Timestamp]  
public class CourseController : SocratesController
```


6. **Run** the application and check the results when viewing the list of courses.




 Now, we'll register the **TimestampAttribute** as a **global filter** so that it will be in effect for all of the application's controllers and actions.

7. Remove the Timestamp attribute from CourseController and register the attribute as a **global filter** in the **RegisterGlobalFilters** method within **FilterConfig.cs**. You will need a **using** directive for **Socrates.Filters**.

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)  
{  
    filters.Add(new HandleErrorAttribute());  
    filters.Add(new TimestampAttribute());  
}
```

 Note that if you apply the filter to a controller and have it registered as a global filter, it will only be applied once.

8. **Run** the application and confirm that the time values are displayed at the top of every page.
9. **Remove** or **comment-out** the statement that adds Timestamp attribute as a global filter. If you don't, it will interfere with the next lab.

 You have successfully completed Lab 21.


Lab 22

Objectives


- Create a Web API controller for courses
- Retrieve a list of courses in both JSON and XML

Procedures

1. If not already open, re-open your solution from Lab 21.

 If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder of Lab 22.

2. Add a new **folder** to the project named **API**.
3. Right-click on the **API** folder, select [**Add > Controller**], select **Web API 2 Controller - Empty**, and name the controller **CourseController**.

 Notice that we are naming this controller **CourseController** even though we already have a class with that name. It's okay because they are in different **namespaces** and will be accessed via different **routes**.


4. Add a **private variable** to hold an instance of **ISocratesContext**. You will need a using directive for **Socrates.DataAccess**.

```
private ISocratesContext context;
```

5. Override the **Initialize** method and used it to initialize the context.

```
protected override void Initialize(...)
{
    base.Initialize(controllerContext);
    string connStr = ConfigurationManager.ConnectionStrings["ConnStr"].ConnectionString;

    context = SocratesContextFactory.GetContext(connStr, false);
}
```

 Notice that we are disabling the **proxy generation** feature of Entity Framework. This is necessary to avoid a **circular reference** when Web API tries to **serialize** a model object.

6. Add a **Get** action that returns all of the courses.

```
public IEnumerable<Course> Get()
{
    return context.GetAllCourses().ToList();
}
```

7. **Run** the application and change the URL to **http://localhost:[port]/api/course**. If possible, make the same request using a different browser.



If using IE, the response will come back as a file containing JSON data. If using Chrome, the response will come back as XML and display directly in the browser. This behavior is because of what the browsers send for the HTTP **accept header** by default.



You have successfully completed Lab 22.