Certificate in C# Advanced Programming

Creating Web Applications in C#

By Randal Root

# Module 09: MVC and Databases

In this module, we’ll look at how ASP.NET’s Model-View-Controller (MVC) works with databases using the **MVC Data (or Domain) Model**. The MVC Domain Models **often** consists of an Entity Framework Data Model over a SQL Server **Database, but can also be data read from a text file or web service**. This **data model holds a copy of the data in-memory**. Changes to the **data model are changed in memory first** and **later** **changed in the source** either **immediately or asynchronously**. This design involves a complex set of components, but it's currently considered the best practice for creating web applications.

# Working with Databases and ADO.NET

In the last class session, we saw **ADO.NET** accessing and manipulating data in a database. This option is always available and is **a simple way** to add connectivity to a database using code like this:

// GET: Demo1

public ActionResult Index()

{

System.Data.SqlClient.SqlConnection objCon;

System.Data.SqlClient.SqlCommand objCmd;

objCon = new System.Data.SqlClient.SqlConnection(@"Data Source=.;

Integrated Security=SSPI;

Initial Catalog=Northwind");

objCmd = new System.Data.SqlClient.SqlCommand("Select Count(\*) From Orders", objCon);

//using the table directly

objCon.Open();

ViewData["Results"] = objCmd.ExecuteScalar

objCon.Close();

return View();

}//end Index

## Adding Abstraction Layers

While this code works, **connecting directly from a Controller to a table is** **not considered a best practice** in applications. Instead, it is **better to use multiple abstraction layers**. These layers provide for additional flexibility when testing or updating applications at the cost of complexity.

For example, using the Northwind database, we add some code to **create SQL stored procedures and a view that abstract a table data**. After this, we create a **processing layer** of classes and interfaces that **connects our MVC application to the database engine**. Then, we use an MVC **Model** to **hold data extracted by the processing layer**. The MVC **controller processes data from the MVC Model** **to the MVC View**, which is used by humans.

You can visualize this as three layers; Application, Processing, and Data, each containing individual components of abstraction.

MVC **View** <---> MVC **Controller** <---> MVC **Model**

**Processing** **Types** (classes and interfaces)

Database **Stored Procedure** <---> **View** <---> **Table** (Source)

# The Data Layer

The data layer includes simple or complex sets of data. This data can be stored as a software object such as an Enumeration or List of objects, or it can be stored as text in one or more text files. However, as the data gets more complex, it is more often stored in a set of relational database tables.

DROP TABLE IF EXISTS Students;

CREATE TABLE [Students](

[StudentId] [int] NOT NULL,

[StudentName] [nvarchar](100) NOT NULL,

[StudentEmail] [nvarchar](100) NOT NULL,

[StudentLogin] [nvarchar](50) NULL,

[StudentPassword] [nvarchar](50) NULL,

CONSTRAINT [PK\_Students] PRIMARY KEY CLUSTERED

( [StudentId] ASC )

)

Go

## Database Views

These table are abstracted by views and stored procedures. Each table should include a basic, or base, view that represents the table as a whole.

DROP VIEW IF EXISTS vStudents;

Go

Create View vStudents

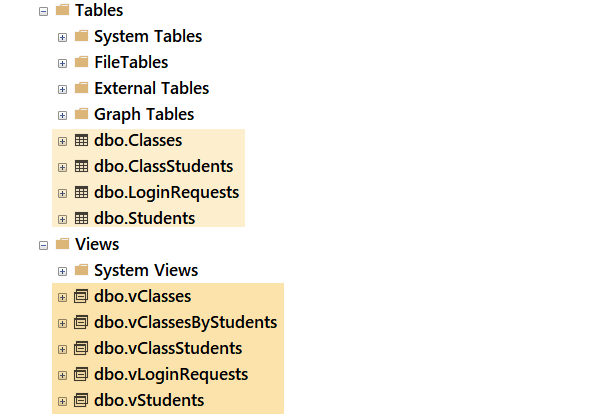
AS

Select [StudentId], [StudentName], [StudentEmail], [StudentLogin], [StudentPassword] From [Students]

Go

The view is used when data from that table needs to be selected.

string strCmd = @"Select StudentID, StudentName, StudentEmail, StudentLogin, StudentPassword From vStudents;";



## Database Stored Procedures

Stored procedures abstract the transactions performed on each table. For each type of transaction (insert, update and delete) you would create a corresponding store procedure.

DROP PROCEDURE IF EXISTS pInsStudents;

Go

Go

CREATE

PROCEDURE pInsStudents (

@StudentId int

, @StudentName nvarchar(100)

, @StudentEmail nvarchar(100)

, @StudentLogin nvarchar(50)

, @StudentPassword nvarchar(50)

)

AS

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Desc: This stored procedure adds data to the Students table

ChangeLog: (Who, When, What)

RRoot, 2030.01.01, created procedure

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BEGIN -- Body of stored procedure:

BEGIN TRY

BEGIN TRANSACTION;

------------------- Transaction Statement:

INSERT INTO [Students]

([StudentId], [StudentName], [StudentEmail], [StudentLogin], [StudentPassword])

VALUES(@StudentId, @StudentName, @StudentEmail, @StudentLogin, @StudentPassword);

------------------- Transaction Statement;

COMMIT TRANSACTION;

RETURN +100

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION;

Declare @Message nVarchar(1000);

Select @Message = ERROR\_MESSAGE();

RAISERROR(@Message, 15, 1);

RETURN -100

END CATCH

END; -- Body of stored procedure;

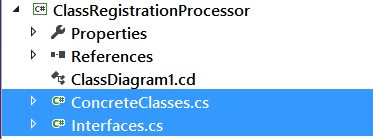
Go

**Note**: It is often good to include a procedure of Selecting data as well since it makes it easy to access a single row of data using a parameter.

# The Processing Layer

We create code to process data from the database to the application and back again in the processing layer. This layer often has its own abstraction layer components in the form of a set of interfaces and their concrete classes.

Although the processing layer code can be included within an application, it is often better to create a separate project that can be reused by multiple applications. These projects would include both the interface and the classes that implement them.



## Interfaces

Interface define the pattern of a Structure or Class. They are used to like a blueprint is used to create a building. A blueprint defines what must be included in the building, although more things can be added if the builder gets approval to do so. In a similar way, interfaces define what must be included in the Structure or Class (although more things can be added if it is approved by the development team.) In our example, we define the data needed to record and process student information. We define one interface to record student data and another to process that data.

public interface iStudent

{

int StudentID { get; set; }

string StudentName { get; set; }

string StudentEmail { get; set; }

string StudentLogin { get; set; }

string StudentPassword { get; set; }

}

//Processors

public interface iStudentProcessor

{

List<Student> Select(string Connection);

int Insert(string Connection, int StudentID, string StudentName,

string StudentEmail, string StudentLogin, string StudentPassword);

int Update(string Connection, int StudentID, string StudentName,

string StudentEmail, string StudentLogin, string StudentPassword);

int Delete(string Connection, int StudentID);

}

## Classes

Classes define and track data within an application, as well as perform actions on that data. In our example, we ant to define and track student data.

public class Student : iStudent

{

//Can be replaced with Invisible Auto Property fields

private int \_StudentID;

private string \_StudentName;

private string \_StudentEmail;

private string \_StudentLogin;

private string \_StudentPassword;

//Constructor

public Student(int StudentID, string StudentName, string StudentEmail, string StudentLogin, string StudentPassword)

{

this.StudentID = StudentID;

this.StudentName = StudentName;

this.StudentEmail = StudentEmail;

this.StudentLogin = StudentLogin;

this.StudentPassword = StudentPassword;

}

public int StudentID { get => \_StudentID; set => \_StudentID = value; }

public string StudentName { get => \_StudentName; set => \_StudentName = value; }

public string StudentEmail { get => \_StudentEmail; set => \_StudentEmail = value; }

public string StudentLogin { get => \_StudentLogin; set => \_StudentLogin = value; }

public string StudentPassword { get => \_StudentPassword; set => \_StudentPassword = value; }

public override string ToString()

{

return this.StudentID + ","

+ StudentName + ","

+ StudentEmail + ","

+ StudentLogin + ","

+ StudentPassword + ";";

}

}

public class StudentProcessor : iStudentProcessor

{

public List<Student> Select(string ConnectionString)

{

try

{

string strCmd = @"Select StudentID, StudentName, StudentEmail, StudentLogin, StudentPassword From vStudents;";

SqlConnection objCon = new SqlConnection(ConnectionString);

SqlCommand objCmd = new SqlCommand(strCmd, objCon);

objCon.Open();

System.Data.IDataReader objDR = objCmd.ExecuteReader();

List<Student> Students = new List<Student>();

while (objDR.Read())

{

Student objRow = new Student((int)objDR["StudentID"]

, (string)objDR["StudentName"]

, (string)objDR["StudentEmail"]

, (string)objDR["StudentLogin"]

, (string)objDR["StudentPassword"]);

Students.Add(objRow);

}

objCon.Close();

return Students;

}

catch (Exception)

{ throw; }

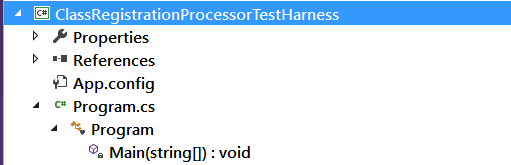
}

//Add other methods to implement iStudentProcessor

}

## Testing the Processing Layer

One of the advantages of using a separate processing layer is that multiple applications can use its processing code, this allows it to be tested separate from the application layer. You do this by creating a testharness program and verifying that each method executes correctly.



class Program

{

static void Main(string[] args)

{

string strCon = "Data Source=.;Initial Catalog=AdvWebDevProject;Integrated Security=True";

ClassRegistrationProcessor.StudentProcessor objSP;

objSP = new ClassRegistrationProcessor.StudentProcessor();

List<ClassRegistrationProcessor.Student> objStudents;

int intRC;

Console.WriteLine("Select Test");

objStudents = objSP.Select(strCon);

foreach (var item in objStudents) Console.WriteLine(item.ToString());

Console.WriteLine("\n\r");

Console.WriteLine("Insert Test");

intRC = objSP.Insert(strCon, 3, "Test I", "Test I", "Test I", "Test I");

Console.WriteLine(intRC);

objStudents = objSP.Select(strCon);

foreach (var item in objStudents) Console.WriteLine(item.ToString());

Console.WriteLine("\n\r");

Console.WriteLine("Update Test");

intRC = objSP.Update(strCon, 3, "Test U", "Test U", "Test U", "Test U");

Console.WriteLine(intRC);

objStudents = objSP.Select(strCon);

foreach (var item in objStudents) Console.WriteLine(item.ToString());

Console.WriteLine("\n\r");

Console.WriteLine("Delete Test");

intRC = objSP.Delete(strCon, 3);

Console.WriteLine(intRC);

foreach (var item in objStudents) Console.WriteLine(item.ToString());

Console.WriteLine("\n\r");

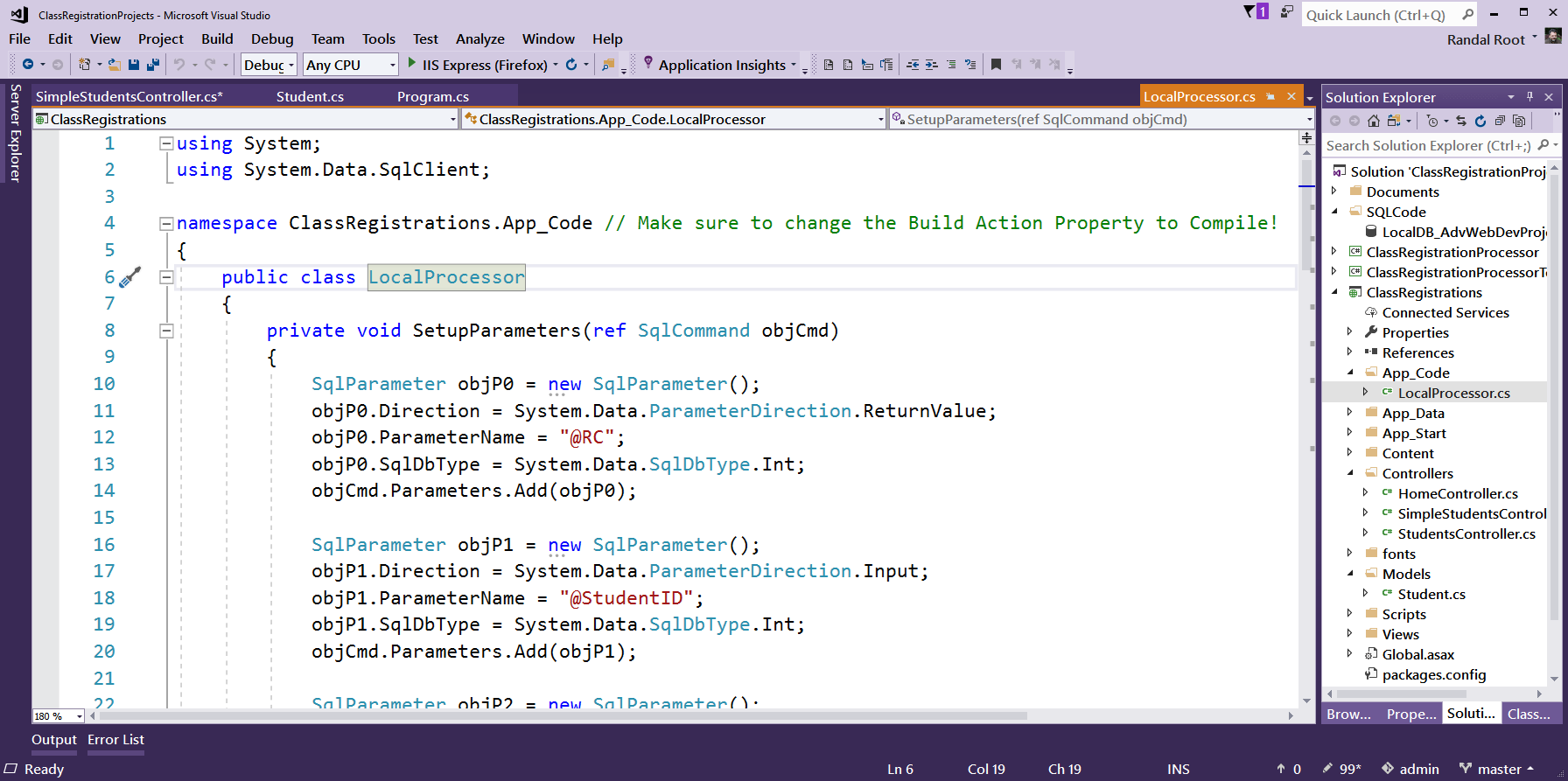
Console.ReadLine();//Pause

}

}

## Including the Processing Layer in an MVC Application

You can optionally place the processing layer code within an MVC application, app\_code folder. Using this option blocks other applications from using your code. Of course, this means that it does not allow you to test the code outside of the MVC application either.



**NOTE:** Make sure to change the **Build Action property to Compile** on the C# code file when using an App\_Code folder in an MVC application!

# The Applications Layer

In an MVC application, you work with a set of models, views, and controllers. These components separate the

"Model

The **central component of the pattern**. It is the application's dynamic data structure, **independent of the user interface**.[[4]](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#cite_note-4) It directly manages the data, logic and rules of the application.

View

**Any representation of information** such as a chart, diagram or table. **Multiple views of the same information** are possible, such as a bar chart for management and a tabular view for accountants.

Controller

**Accepts input and converts it to commands for the model or view**.[[5]](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#cite_note-5)

([https://en.wikipedia.org/wiki/Model–view–controller](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller), 2019)

## The MVC Data Model

The MVC data model defines the data within an MVC application. In ASP.NET, this implemented as a set of classes, structures, or enumerations. Interfaces can be used, as in the processing layer, but are not required in our example.

public class Student

{

private int \_StudentID;

private string \_StudentName;

private string \_StudentEmail;

private string \_StudentLogin;

private string \_StudentPassword;

//Constructor

public Student(int StudentID, string StudentName, string StudentEmail, string StudentLogin, string StudentPassword)

{

this.StudentID = StudentID;

this.StudentName = StudentName;

this.StudentEmail = StudentEmail;

this.StudentLogin = StudentLogin;

this.StudentPassword = StudentPassword;

}

public int StudentID { get => \_StudentID; set => \_StudentID = value; }

public string StudentName { get => \_StudentName; set => \_StudentName = value; }

public string StudentEmail { get => \_StudentEmail; set => \_StudentEmail = value; }

public string StudentLogin { get => \_StudentLogin; set => \_StudentLogin = value; }

public string StudentPassword { get => \_StudentPassword; set => \_StudentPassword = value; }

}

**Note:** The MVC data model is a local representation of the data that processed in the processing layer. While not required, it can be **useful when you do not have permission to modify or extend code in the processing layer**.

## The MVC Controller

The MCV controller allows you to **control the flow of data** between the processing layer objects and the MCV views. Remember that **processing layers objects can be in both the MVC application and in external components**, such as a C# class library.

Here is a **simple version** of an MVC controller **using the external processor**.

public class DemoController : Controller

{

//Create a Data Processor object

private ClassRegistrationProcessor.StudentProcessor objProcessor;

private string strConnectionString;

public DemoController()

{

objProcessor = new ClassRegistrationProcessor.StudentProcessor();

strConnectionString = "Data Source=(local);Initial Catalog=AdvWebDevProject;Integrated Security=True";

}

private List<ClassRegistrationProcessor.Student> SelectAllStudents()

{

return objProcessor.Select(strConnectionString);

}

private ClassRegistrationProcessor.Student SelectOneStudents(int id)

{

var objOneStudent = new object();

foreach (var item in SelectAllStudents())

{

if (item.StudentID == id) objOneStudent = item;

}

return (ClassRegistrationProcessor.Student)objOneStudent;

}

// GET: DEmo

public ActionResult Index()

{

return View(SelectOneStudents(1));

}

}

Here is another **simple version** of an MVC controller **using an application data module and an application processor**.

// GET: SimpleStudents

public ActionResult Index()

{

List<Models.Student> Students = new List<Models.Student>();

string strConnectionString = System.Configuration.ConfigurationManager.ConnectionStrings["UWSvr"].ConnectionString;

try

{

string strCmd = @"Select StudentID, StudentName, StudentEmail, StudentLogin, StudentPassword From vStudents;";

SqlConnection objCon = new SqlConnection(strConnectionString);

SqlCommand objCmd = new SqlCommand(strCmd, objCon);

objCon.Open();

System.Data.IDataReader objDR = objCmd.ExecuteReader();

while (objDR.Read())

{

Models.Student objRow = new Models.Student((int)objDR["StudentID"]

, (string)objDR["StudentName"]

, (string)objDR["StudentEmail"]

, (string)objDR["StudentLogin"]

, (string)objDR["StudentPassword"]);

Students.Add(objRow);

}

objCon.Close();

}

catch (Exception)

{

throw;

}

return View(Students);

}

#region Insert

// GET: Students/Create

public ActionResult Create() //AKA Insert!

{

return View(); //Open a Razor page with textboxes to fill in new data

}

// POST: Students/Create

[HttpPost]

public ActionResult Create(FormCollection collection) //catch the new data from the textboxes

{

string strConnectionString = System.Configuration.ConfigurationManager.ConnectionStrings["UWSvr"].ConnectionString;

ClassRegistrations.App\_Code.LocalProcessor objProcessor; // Make sure to change class's Build Action Property to Compile!

objProcessor = new App\_Code.LocalProcessor();

// https://amitpatriwala.wordpress.com/2017/07/06/class-file-in-app\_code-folder-not-working-asp-net-mvc/

ViewData["Error"] = ""; //You must declare this here, or it's "Conditonally" created in the Catch block

try

{

objProcessor.Insert(strConnectionString

, int.Parse(collection["StudentID"])

, collection["StudentName"]

, collection["StudentEmail"]

, collection["StudentLogin"]

, collection["StudentPassword"]);

return RedirectToAction("Index");

}

catch (Exception ex)

{

ViewData["Error"] = ex.Message;

return View();

}

}

#endregion

Here is a more complex example where the code uses **an external class library for its processing layer**. Note how **you use the processing layer data models to get the data and then hand it off to the application data model for presentation**. Doing this makes it easier to change to different processing layer object in the future.

//Create a Data Processor object

private ClassRegistrationProcessor.StudentProcessor objProcessor;

private string strConnectionString;

public StudentsController()

{

objProcessor = new ClassRegistrationProcessor.StudentProcessor();

//strConnectionString = "Data Source=(local);Initial Catalog=AdvWebDevProject;Integrated Security=True";

strConnectionString = System.Configuration.ConfigurationManager.ConnectionStrings["LocalPC"].ConnectionString;

strConnectionString = System.Configuration.ConfigurationManager.ConnectionStrings["UWSvr"].ConnectionString;

}

private List<Models.Student> SelectAllStudents()

{

//You can return an object for View data using a class in the DLL,

List<ClassRegistrationProcessor.Student> lstRows = objProcessor.Select(strConnectionString);

//BUT you can also create an MVC data model, which decouples the requierment of a DLL (could switch to a XML or JSON file later!)

Models.Student objStudent;

List<Models.Student> lstStudents = new List<Models.Student>();

foreach (var row in lstRows)

{

objStudent = new Models.Student(row.StudentID, row.StudentName, row.StudentEmail, row.StudentLogin, row.StudentPassword);

lstStudents.Add(objStudent);

}

return lstStudents;

}

private Models.Student SelectOneStudents(int id)

{

var objOneStudent = new object();

foreach (var item in SelectAllStudents())

{

if (item.StudentID == id) objOneStudent = item;

}

return (Models.Student)objOneStudent;

}

//GET: Students

public ActionResult Index()

{

return View(SelectAllStudents()); //Send a list of student objects to the Razor page

}

// GET: Students/Details/5

public ActionResult Details(int id)

{

return View(SelectOneStudents(id)); //Send a student object to the Razor page

}

## The MVC Views

MVC Views present and gather data to and from a user. **By returning a reference to a data model you can access easily build a user interface with Razor code**. Notice that the **module type returned from the controller must be of the same type as referenced in the view**.

@model IEnumerable<ClassRegistrations.Models.Student>

@{ Layout = null;}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Index</title>

</head>

<body>

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

<table class="table">

<tr>

<th>StudentName</th>

<th>StudentEmail</th>

<th>StudentLogin</th>

<th>StudentPassword</th>

</tr>

@foreach (var item in Model)

{

<tr>

<td>@item.StudentName</td>

<td>@item.StudentEmail</td>

<td>@item.StudentLogin</td>

<td>@item.StudentPassword</td>

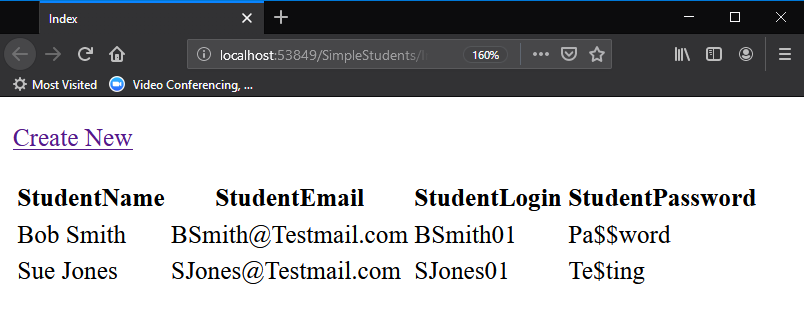
</tr>

}

</table>

</body>

</html>



## Linking Views

Notice the **action link in the View** that points to a "Create" controller method. This link **allows you to link one view to another**. In this case, the **new** **view gathers data** **from a user** **which is later inserted into the data layer** database. This **involves two controller methods**, **one to gather** the data from the user and **one to submit** that data to the processing layer.

Here is an example of the controller first controller.

public ActionResult Create()

{

return View();

}

This method **does not pass an object filled with data**, **so** there are **no arguments used** in the call to the View() method. From here, we can create a simple page using standard HTML code. However, **if** we want to have **Razor** help us build the correct HTML form, we need to **identify the model** will use in view with the "@model" declaration. **Using HTML helpers via Razor is optional**, but a "nice to have" option!

@model ClassRegistrations.Models.Student

@{ Layout = null; }

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Create</title>

</head>

<body>

@using (Html.BeginForm())

{

<h4>Add New Student</h4>

<div>

<span>Student ID</span> @Html.TextBox("StudentID")<br />

<span>Student Name</span> @Html.TextBox("StudentName")<br />

<span>Student Email</span> @Html.TextBox("StudentEmail")<br />

<span>Student Login</span> @Html.TextBox("StudentLogin")<br />

<span>Student Password</span> @Html.TextBox("StudentPassword")<br />

<input type="submit" value="Insert" />

</div>

}

<div>

@Html.ActionLink("Back to List", "Index")

</div>

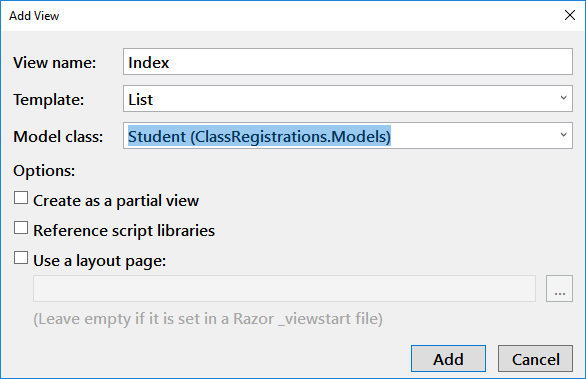
</body>

</html>

## Using MVC View Templates

One of the features students like in ASP.NET MVC projects is its ability to create **lots of free code** using **built-in templates**. While these **create a utilitarian user interface**, it is an **easy way to test your controller code**.

**Note**: When using the List scaffold, Visual Studio **assumes** you are working with an **IEnumerable** sequence of the model view type, so you can just **select the singular** form of the class from the list



When you use the template, a lot of code will be created for you. Here is an example of a "List" template's code:

@model IEnumerable<ClassRegistrations.Models.Student>

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

</p>

<table class="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.StudentID)

</th>

<th>

@Html.DisplayNameFor(model => model.StudentName)

</th>

<th>

@Html.DisplayNameFor(model => model.StudentEmail)

</th>

<th>

@Html.DisplayNameFor(model => model.StudentLogin)

</th>

<th>

@Html.DisplayNameFor(model => model.StudentPassword)

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.StudentID)

</td>

<td>

@Html.DisplayFor(modelItem => item.StudentName)

</td>

<td>

@Html.DisplayFor(modelItem => item.StudentEmail)

</td>

<td>

@Html.DisplayFor(modelItem => item.StudentLogin)

</td>

<td>

@Html.DisplayFor(modelItem => item.StudentPassword)

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id = item.StudentID }) |

@Html.ActionLink("Details", "Details", new { id = item.StudentID }) |

@Html.ActionLink("Delete", "Delete", new { id = item.StudentID })

</td>

</tr>

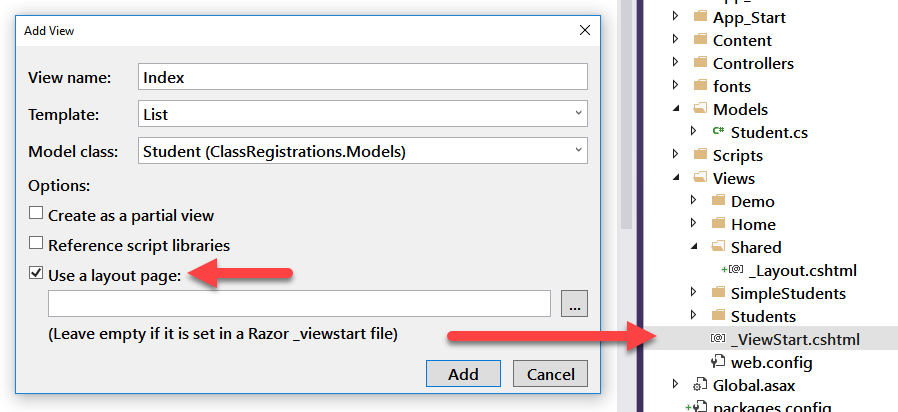
}

</table>

Note that the template includes several ActionLinks, but these are not working yet. It is up to you to create the controllers and views complete their actions.

## Adding the Layout View

When you use the built-in templates, the dialog window includes an option to use a layout page. If you check this checkbox. and leave its associated textbox empty, a new layout page will be created for you and referenced in the "\_ViewStart.cshtml" file.



"An application may **contain common parts in the UI which remains the same throughout the application** such as the logo, header, left navigation bar, right bar or footer section. ASP.NET MVC introduced a Layout view which contains these common UI parts, so that we don't have to write the same code in every page. The **layout view is same as the master page of the ASP.NET webform** application." (<https://www.tutorialsteacher.com/mvc/layout-view-in-asp.net-mvc>, 2019)

# Lab 01: An Instructor Led Demo

In this lab, you will follow along with your instructor as they walk you though an example ASP.NET MVC application.