Controllers in ASP.NET MVC

In this section, you will learn about the Controller in ASP.NET MVC.

The Controller in MVC architecture handles any incoming URL request. The Controller is a class, derived from the base class System.Web.Mvc.Controller. Controller class contains public methods called **Action** methods. Controller and its action method handles incoming browser requests, retrieves necessary model data and returns appropriate responses.

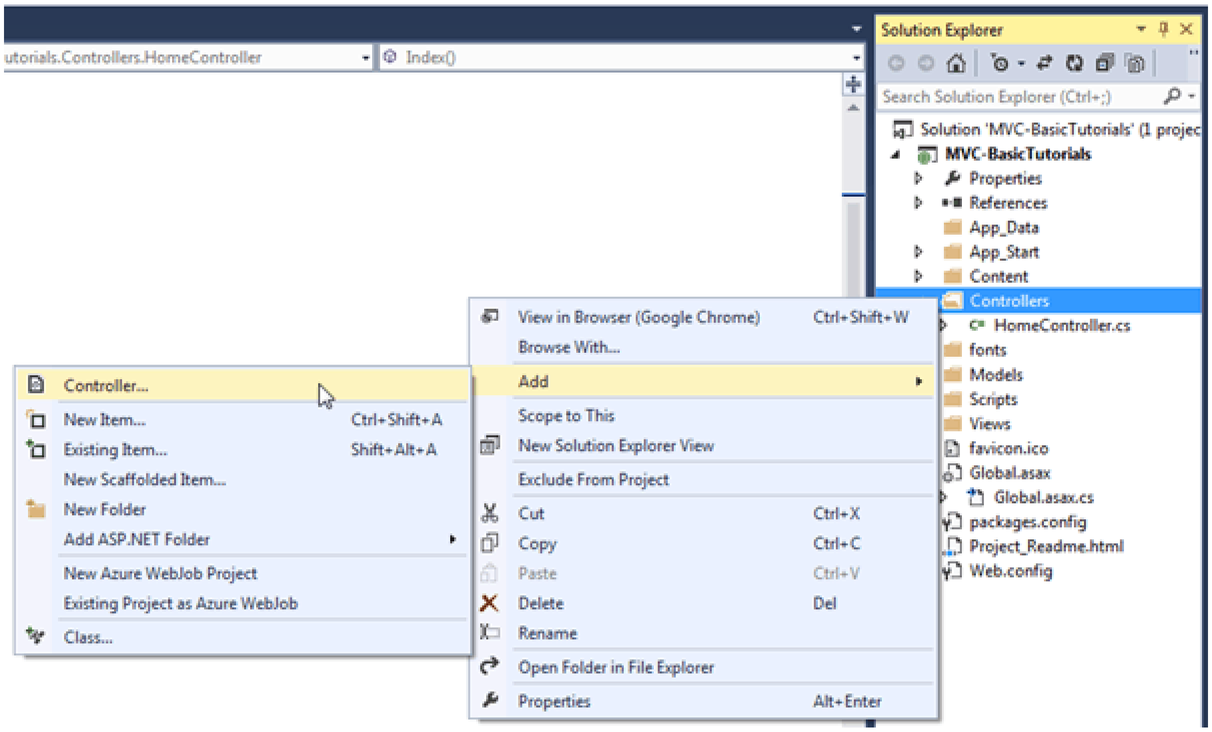
In ASP.NET MVC, every controller class name must end with a word "Controller". For example, the home page controller name must be HomeController, and for the student page, it must be the StudentController. Also, every controller class must be located in the Controller folder of the MVC folder structure.

Adding a New Controller

Now, let's add a new empty controller in our MVC application in Visual Studio.

In the previous section, we learned how to create our first MVC application, which created a default HomeController. Here, we will create new StudentController class.

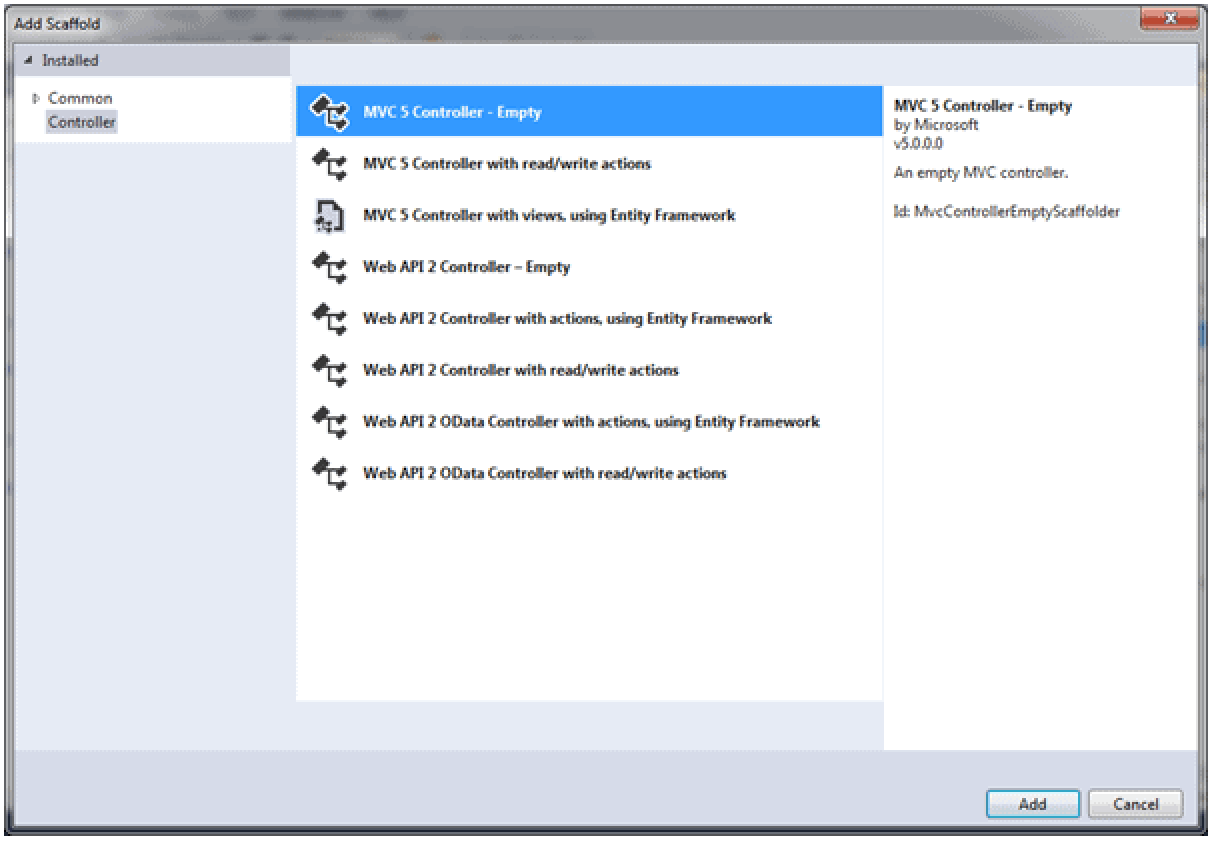
In the Visual Studio, right click on the Controller folder -> select **Add** -> click on **Controller.**



Add New Controller

This opens Add Scaffold dialog, as shown below.

Scaffolding is an automatic code generation framework for ASP.NET web applications. Scaffolding reduces the time taken to develop a controller, view, etc. in the MVC framework. You can develop a customized scaffolding template using T4 templates as per your architecture and coding standards.

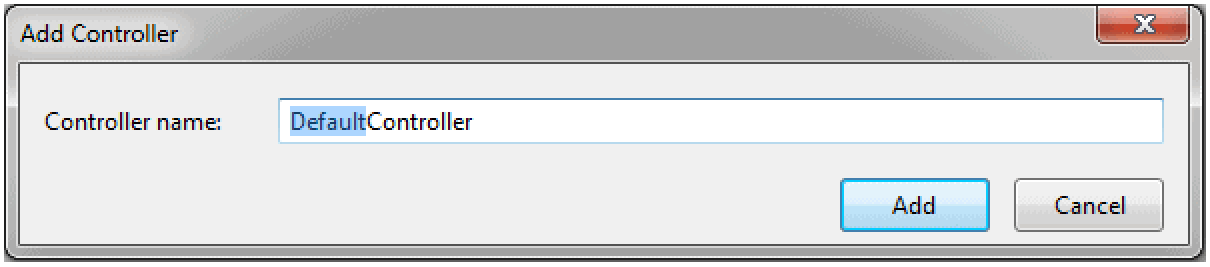

Add Scaffold

Installed
- Common
- Controller

MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData Controller with actions, using Entity Framework
Web API 2 OData Controller with read/write actions

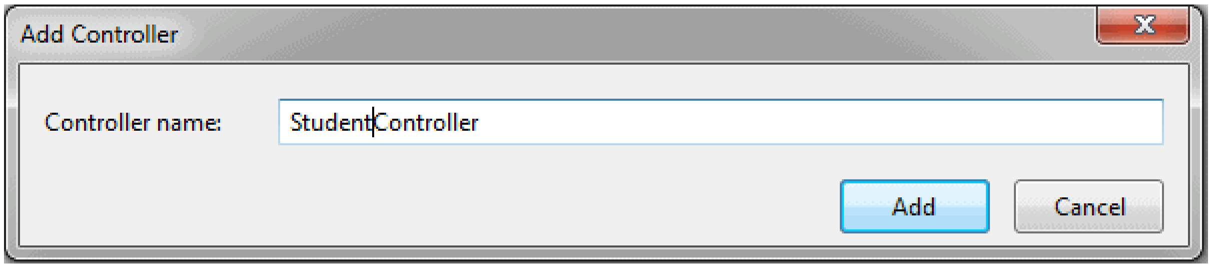
MVC 5 Controller - Empty
by Microsoft
v5.0.0.0
An empty MVC controller.
Id: MvcControllerEmptyScaffolder

Add Scaffold dialog contains different templates to create a new controller. We will learn about other templates later. For now, select "MVC 5 Controller - Empty" and click Add. It will open the Add Controller dialog, as shown below


Add Controller

Controller name: DefaultController

In the **Add Controller** dialog, enter the name of the controller. Remember, the controller name must end with Controller. Write StudentController and click **Add**.



Controller name: StudentController

This will create the StudentController class with the Index() method in StudentController.cs file under the Controllers folder, as shown below.

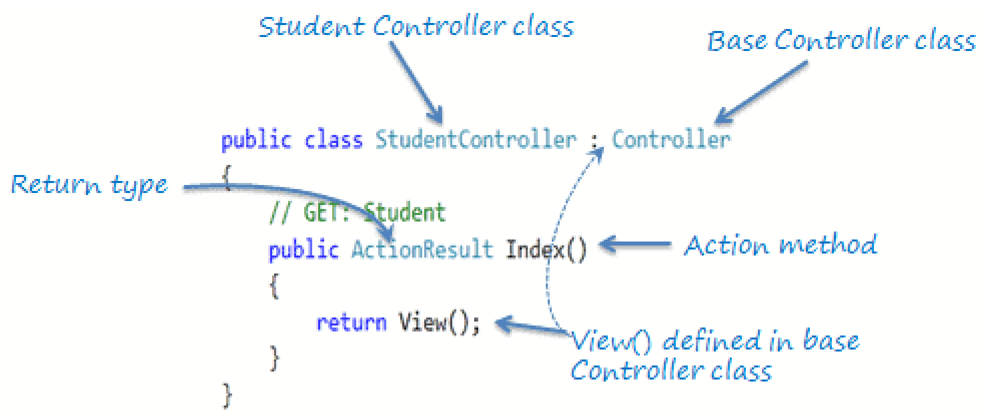
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Action method

All the public methods of the `Controller` class are called `Action` methods. They are like any other normal methods with the following restrictions:

1. Action method must be public. It cannot be private or protected
2. Action method cannot be overloaded
3. Action method cannot be a static method.

The following illustrates the `Index()` action method in the `StudentController` class.



```
public class StudentController : Controller
{
    // GET: Student
    public ActionResult Index()
    {
        return View();
    }
}
```

Action Method

As you can see in the above figure, the `Index()` method is public, and it returns the `ActionResult` using the `View()` method. The `View()` method is defined in the `Controller` base class, which returns the appropriate `ActionResult`.

ActionResult

MVC framework includes various `Result` classes, which can be returned from an action method. The result classes represent different types of responses, such as HTML, file, string, JSON, javascript, etc. The following table lists all the result classes available in ASP.NET MVC.

Result Class	Description
ViewResult	Represents HTML and markup.
EmptyResult	Represents No response.
ContentResult	Represents string literal.
FileContentResult/ FilePathResult/ FileStreamResult	Represents the content of a file.
JavaScriptResult	Represent a JavaScript script.
JsonResult	Represent JSON that can be used in AJAX.
RedirectResult	Represents a redirection to a new URL.
RedirectToRouteResult	Represent another action of same or other controller.
PartialViewResult	Returns HTML from Partial view.
HttpUnauthorizedResult	Returns HTTP 403 status.

The `ActionResult` class is a base class of all the above result classes, so it can be the return type of action method that returns any result listed above. However, you can specify the appropriate result class as a return type of action method.

The `Index()` method of the `StudentController` in the above figure uses the `View()` method to return a `ViewResult` (which is derived from the `ActionResult` class). The base `Controller` class includes the `View()` method along with other methods that return a particular type of result, as shown in the below table.

Result Class	**Description**	**Base Controller Method**
ViewResult	Represents HTML and markup.	View()
EmptyResult	Represents No response.	
ContentResult	Represents string literal.	Content()
FileContentResult, FilePathResult, FileStreamResult	Represents the content of a file.	File()
JavaScriptResult	Represents a JavaScript script.	JavaScript()
JsonResult	Represents JSON that can be used in AJAX.	Json()
RedirectResult	Represents a redirection to a new URL.	Redirect()
RedirectToRouteResult	Represents redirection to another route.	RedirectToRoute()
PartialViewResult	Represents the partial view result.	PartialView()
HttpUnauthorizedResult	Represents HTTP 403 response.	

As you can see in the above table, the View() method returns the ViewResult, the Content() method returns a string, the File() method returns the content of a file, and so on. Use different methods mentioned in the above table to return a different type of result from an action method.

Action Method Parameters

Every action methods can have input parameters as normal methods. It can be primitive data type or complex type parameters, as shown below.

Example: Action Method Parameters

```
[HttpPost]
public ActionResult Edit(Student std)
{
    // update student to the database

    return RedirectToAction("Index");
}

[HttpDelete]
public ActionResult Delete(int id)
{
    // delete student from the database whose id matches with specified
id

    return RedirectToAction("Index");
}
Please note that action method paramter can be Nullable Type.
```

By default, the values for action method parameters are retrieved from the request's data collection. The data collection includes name/values pairs for form data or query string values or cookie values. Model binding in ASP.NET MVC automatically maps the URL query string or form data collection to the action method parameters if both names match.

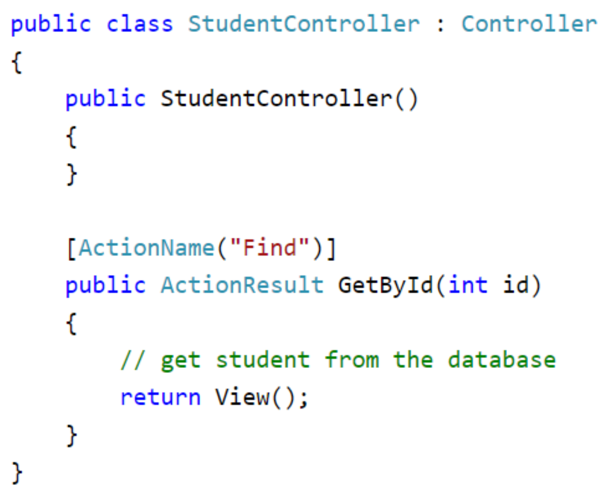
Action Selectors

Action selector is the attribute that can be applied to the action methods. It helps the routing engine to select the correct action method to handle a particular request. MVC 5 includes the following action selector attributes:

1. ActionName
2. NonAction
3. ActionVerbs

ActionName

The `ActionName` attribute allows us to specify a different action name than the method name, as shown below.

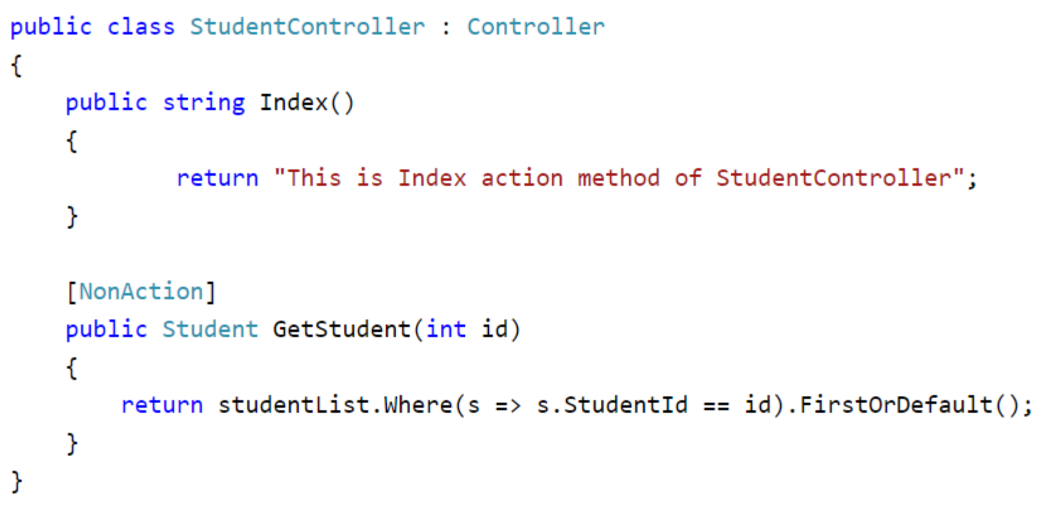


```
public class StudentController : Controller
{
    public StudentController()
    {
    }

    [ActionName("Find")]
    public ActionResult GetById(int id)
    {
        // get student from the database
        return View();
    }
}
```

In the above example, we have applied ActioName("find") attribute to the GetById() action method. So now, the action method name is Find instead of the GetById. So now, it will be invoked on

http://localhost/student/find/1 request instead of http://localhost/student/getbyid/1 request.

Non Action



```
public class StudentController : Controller
{
    public string Index()
    {
            return "This is Index action method of StudentController";
    }

    [NonAction]
    public Student GetStudent(int id)
    {
        return studentList.Where(s => s.StudentId == id).FirstOrDefault();
    }
}
```

Use the `NonAction` attribute when you want public method in a controller but do not want to treat it as an action method.

In the following example, the `Index()` method is an action method, but the `GetStudent()` is not an action method.

Action Verbs

ActionVerbs: HttpGet, HttpPost, HttpPut: The `ActionVerbs` selector is to handle different type of Http requests. The MVC framework includes HttpGet, HttpPost, HttpPut, HttpDelete, HttpOptions, and HttpPatch action verbs. You can apply one or more action verbs to an action method to handle different HTTP requests. If you don't apply any action verbs to an action method, then it will handle HttpGet request by default.

The following table lists the usage of HTTP methods:

Http method	Usage
GET	To retrieve the information from the server. Parameters will be appended in the query string.
POST	To create a new resource.
PUT	To update an existing resource.
HEAD	Identical to GET except that server do not return the message body.
OPTIONS	It represents a request for information about the communication options supported by the web server.
DELETE	To delete an existing resource.
PATCH	To full or partial update the resource.

Example:

```
public class StudentController : Controller
{
    public ActionResult Index() // handles GET requests by default
    {
        return View();
    }

    [HttpPost]
    public ActionResult PostAction() // handles POST requests by
default
    {
        return View("Index");
    }


    [HttpPut]
    public ActionResult PutAction() // handles PUT requests by default
    {
        return View("Index");
    }


    [HttpDelete]
```

```
        public ActionResult DeleteAction() // handles DELETE requests by
    default
        {
            return View("Index");
        }

        [HttpHead]
        public ActionResult HeadAction() // handles HEAD requests by
    default
        {
            return View("Index");
        }

        [HttpOptions]
        public ActionResult OptionsAction() // handles OPTION requests by
    default
        {
            return View("Index");
        }

        [HttpPatch]
        public ActionResult PatchAction() // handles PATCH requests by
    default
        {
            return View("Index");
        }
}
```

Modals In Asp.net

In this section, you will learn about the model class in ASP.NET MVC framework.

The model classes represents domain-specific data and business logic in the MVC application. It represents the shape of the data as public properties and business logic as methods.

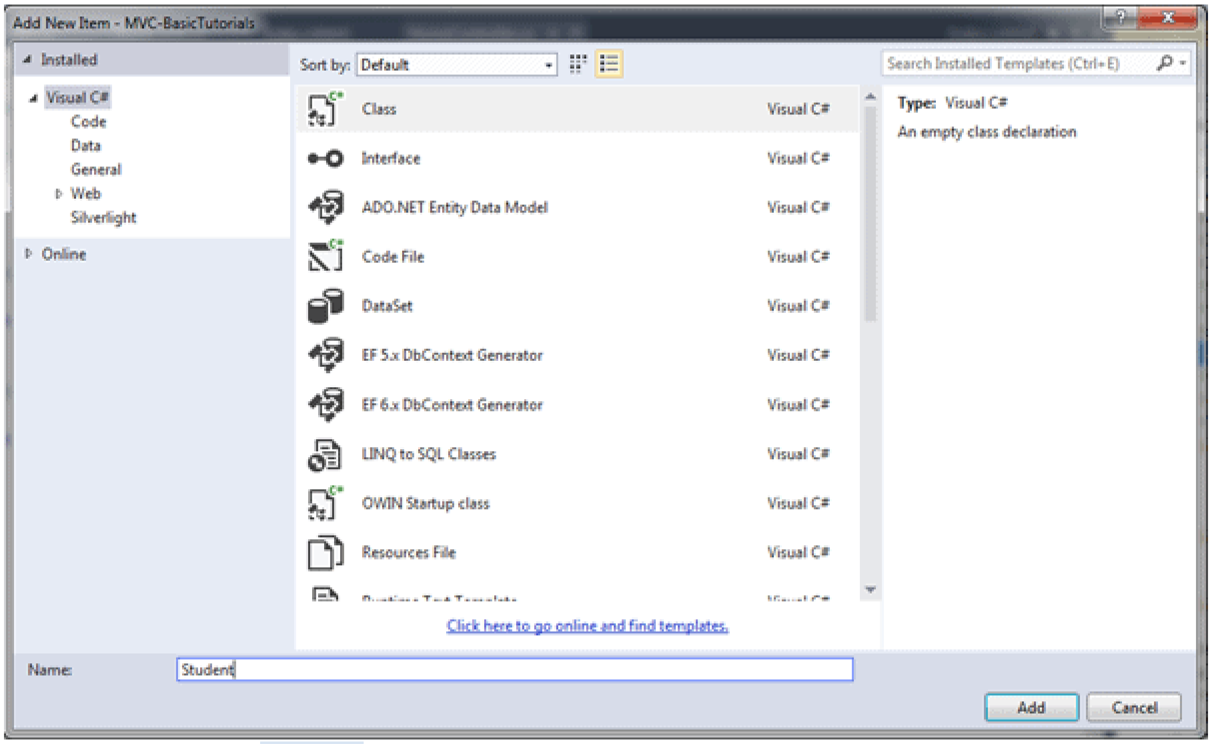
In the ASP.NET MVC Application, all the Model classes must be created in the Model folder.

Adding a Model Class

Let's create the model class that should have the required properties for the `Student` entity.

In the MVC application in Visual Studio, and right-click on the `Model` folder, select **Add** -> and click on **Class...**. It will open the **Add New Item** dialog box.

In the Add New Item dialog box, enter the class name `Student` and click **Add**.


Add New Item - MVC-BasicTutorials
Installed
Visual C#
Code
Data
General
Web
Silverlight
Online
Class
Interface
ADO.NET Entity Data Model
Code File
DataSet
EF 5.x DbContext Generator
EF 6.x DbContext Generator
LINQ to SQL Classes
OWIN Startup class
Resources File
Visual C#
Search Installed Templates (Ctrl+E)
Click here to go online and find templates.
Add
Cancel

Sort by: Default

Type: Visual C#

An empty class declaration

Name: Student

This will add a new `Student` class in model folder. We want this model class to store id, name, and age of the students. So, we will have to add public properties for `Id`, `Name`, and `Age`, as shown below.

Example: Model class

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set;  }
    public int Age { get; set;  }
}
```

The model class can be used in the view to populate the data, as well as sending data to the controller.

Let's create a view and use this model in the next chapter.