

Your professor has created a Visual Studio *project template* that can be used to create new web apps. It includes a fully configured data persistence scheme based on the topics learned in this week's modules. You will continue to use this template as you learn to build interactive ASP.NET MVC web apps.

## Visual Studio 2019

You can download the project template from the course website on Blackboard. Copy the "Web App Project Template V1 - 2xxx.zip" file to the following path on your computer.


**%userprofile%\Documents\Visual Studio 2019\Templates\ProjectTemplates\**

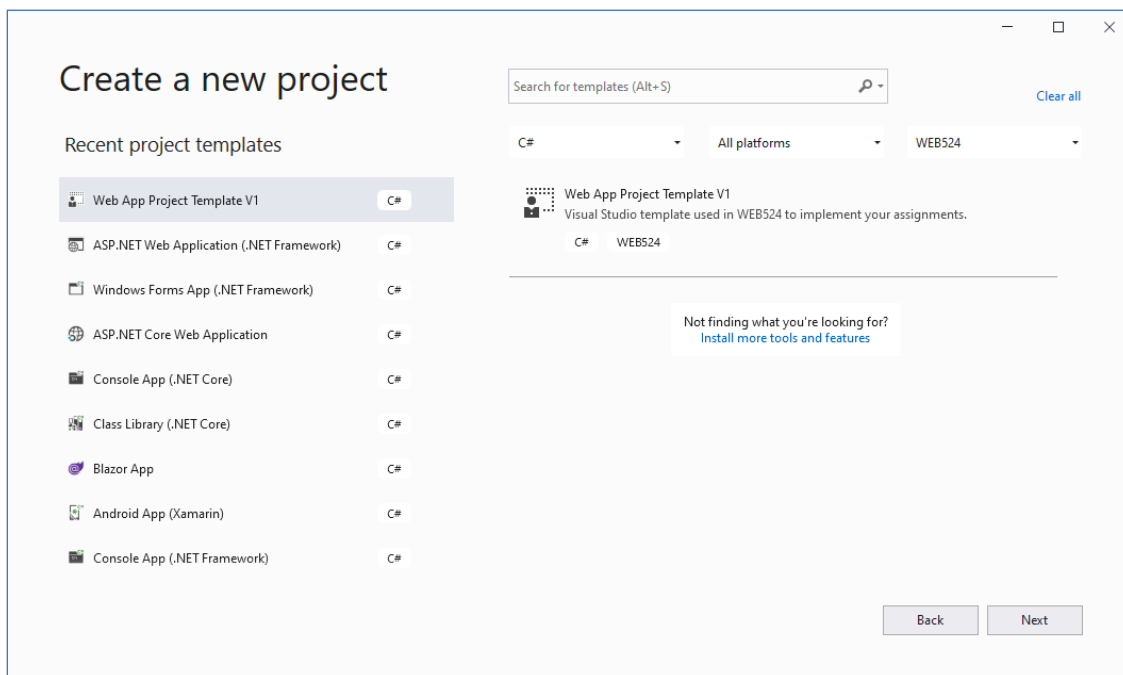
To access the above folder, open Windows *File Explorer*, then from the left panel, click *Documents* under *My Computer*. Then, when you choose File > New > Project in Visual Studio, the template will appear in the center panel, and can be selected to create new web apps.

### Create a new project using the "Web App Project Template V1" template

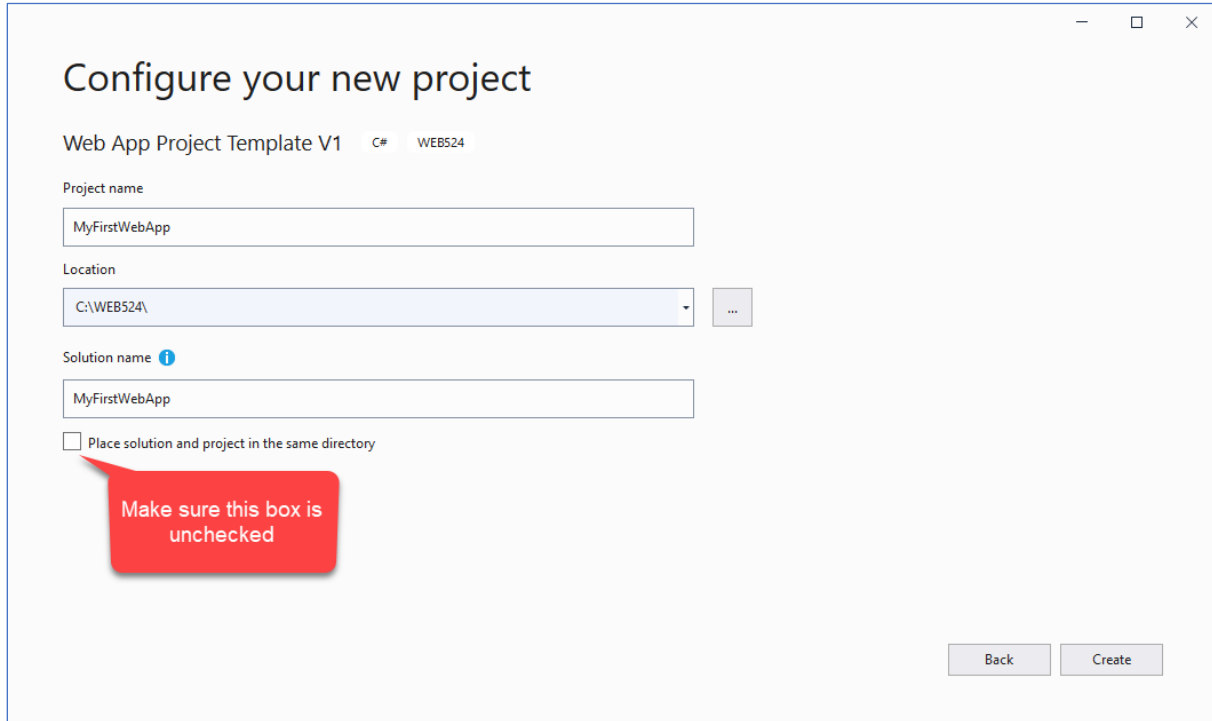
Start Visual Studio and create a new project.

In the "Create a new project" dialog, select "Web App Project Template V1". You may use the filters to narrow down the number of projects. Choose WEB524 in the right-most drop-down. Select the project template then click "Next".

 **Create a new project**  
Choose a project template with code scaffolding to get started



Enter a project name and location then click “Create”.




Configure your new project

Web App Project Template V1 C# WEB524

Project name  
MyFirstWebApp

Location  
C:\WEB524\

Solution name   
MyFirstWebApp

☐ Place solution and project in the same directory

Make sure this box is unchecked

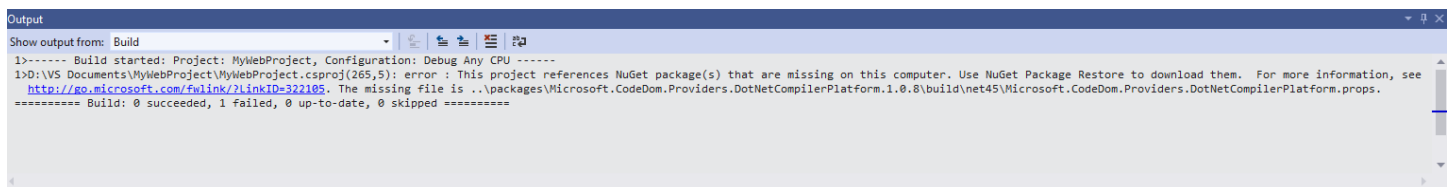
Back Create

After the project loads, build the project so that Visual Studio downloads the necessary packages.

## Error message when building the web app

When you build the project for the first time you may receive the following error in Visual Studio:

*This project references NuGet package(s) that are missing on this computer. Use NuGet Package Restore to download them. For more information, see <http://go.microsoft.com/fwlink/?LinkID=322105>. The missing file is ..\packages\Microsoft.Net.Compilers.1.0.0\build\Microsoft.Net.Compilers.props*



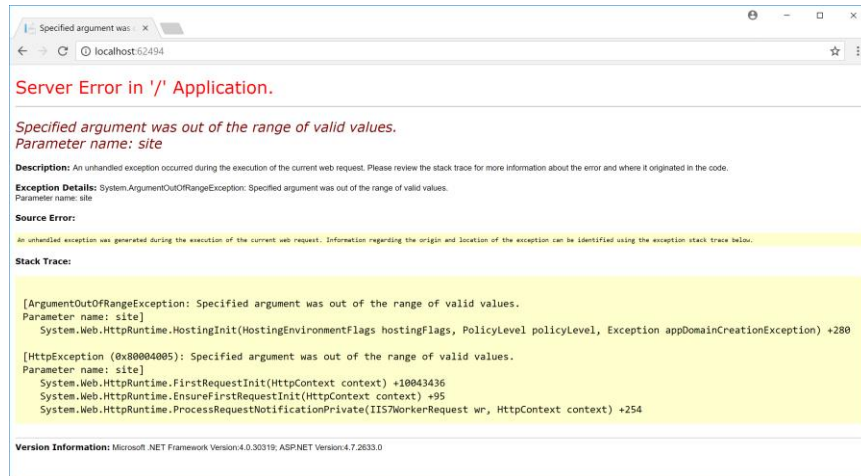
```
Output
Show output from: Build
1>----- Build started: Project: MyWebProject, Configuration: Debug Any CPU -----
1>D:\VS Documents\MyWebProject\MyWebProject.csproj(265,5): error : This project references NuGet package(s) that are missing on this computer. Use NuGet Package Restore to download them. For more information, see
http://go.microsoft.com/fwlink/?LinkID=322105. The missing file is ..\packages\Microsoft.CodeDom.Providers.DotNetCompilerPlatform.1.0.8\build\net45\Microsoft.CodeDom.Providers.DotNetCompilerPlatform.props.
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

If you receive the above error, there is a solution at StackOverflow that seems to do the trick:

<https://stackoverflow.com/a/35101739/923347>

## Error message when running the web app

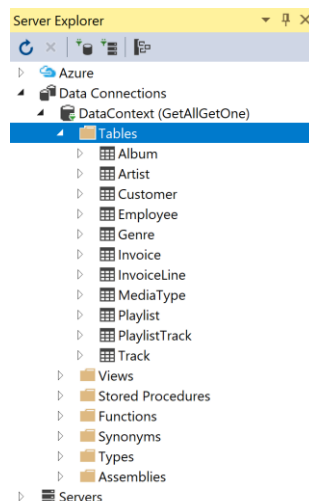
When you start the project for the first time you may receive the following error in your browser. If you do, follow the instructions in the Stack Overflow article [here](#). If your error does not appear to be the same one, ask your professor for help.



Depending on your OS version, when you turn on Internet Information Services you may see a black box, not a checkmark. Do not forget to restart Visual Studio after applying the above fix.

## Explore the Database

In the *Solution Explorer*, double-click the database file **Chinook.mdf** contained in the **App\_Data** folder, a *Server Explorer* window will be displayed.



If the item *DataContext* under the *Data Connections* has a red **✗** on it, it may mean the database needs to be updated to a newer version. To do so, right click *DataContext*, select *Data Connections...*, change nothing and click *OK*.

## Customer “Get All” and “Get One” Functionality

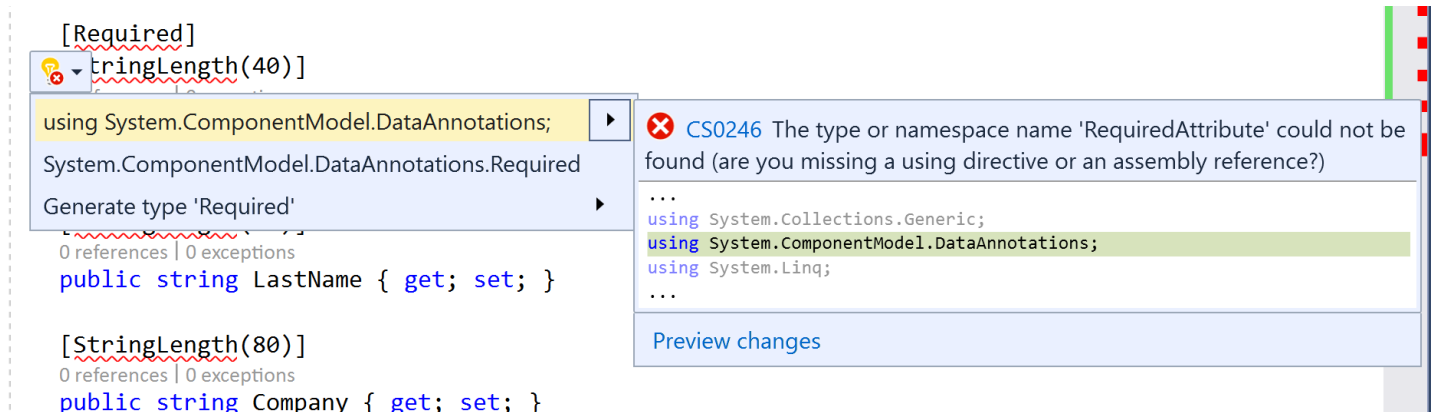
### Create a view model

In the *Models* folder, create a code file to hold a view model class that will describe a *Customer* object.

*Remember* to name the file such that it is the singular word form of the entity or entity group that is being modeled followed by the suffix *ViewModel*. The new code file will contain the *CustomerBaseViewModel* class. As its name suggests, it will have the basic (or base) properties that describe a customer. You can edit the template-provided class name to complete this step.

An easy way to add the properties is to copy them from the *Customer* design model class in the *Data* folder. For this example, we want to copy the properties starting with *CustomerId* up to and including the *Email* property. Ignore (do not copy) the other members.

You will notice that some statements have errors but do not worry, these errors are easily corrected. Simply click on one of the red-underlined statements then press **Ctrl + .** (Control + Period). Visual Studio will analyze the code and suggest some possible changes.

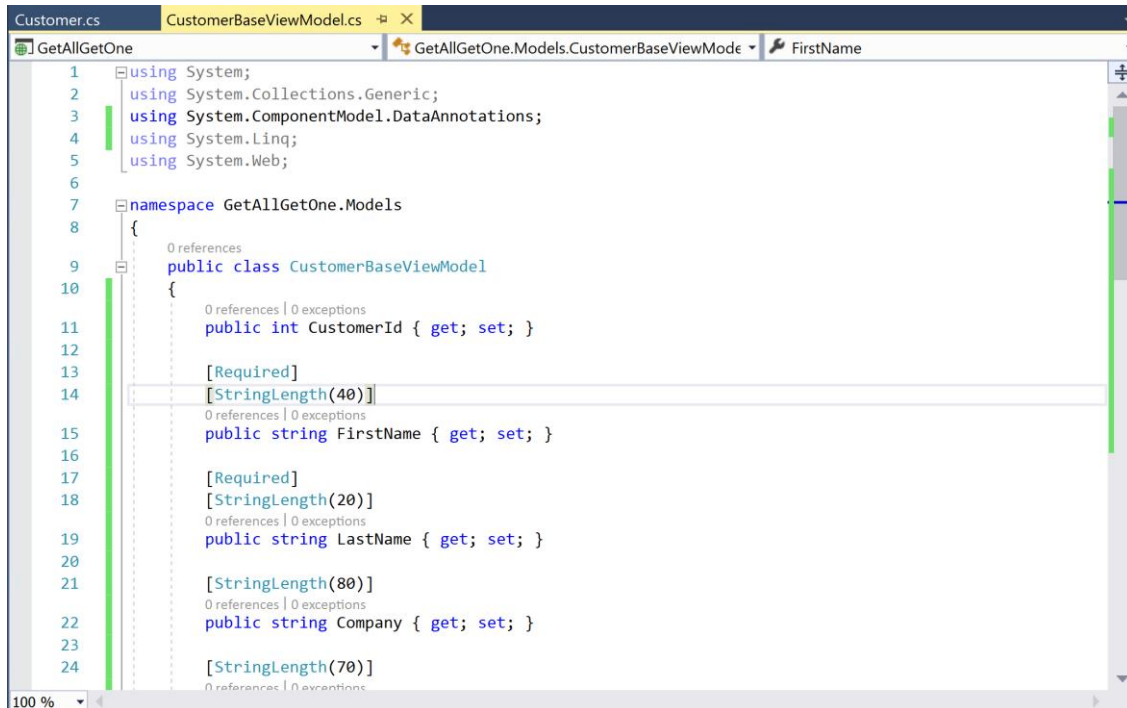


Look at the list of “using” statements at the top of the code file. Notice that VS added the statement that references the data annotations namespace of classes.

The reason for the error message was simply that the source code file did not initially know about the need for data annotations. This technique – Ctrl+. – will be used frequently to enable Visual Studio to help you write better code.

You should have something like the following:

*(continued next page)*

**Primary key:**

What is the primary key of the Customer *design model* class? The *CustomerId* property.

How do we know this? *By convention*, the primary key is an int and its name is either:

**Id**, or **<entityName>Id**, i.e. **CustomerId** for the Customer class

What is the primary key of the CustomerBase *view model* class?

Look for **Id** or **CustomerBaseId**. Did you find it?

No... This will be a problem for the Visual Studio code generators (e.g. view scaffolder). This can be easily fixed by adding a “key” attribute just above the property declaration:

```

[Key]
public int CustomerId { get; set; }

```

## Create a mapper

At this point in time, we have a design model class named *Customer*.

We also have a view model class named *CustomerBaseViewModel*.

We must “map” back-and-forth between design model and view model objects. Why? Remember the dominant system architecture rule: We NEVER allow user-interaction code in controllers to get access to the app’s design model. We ALWAYS use a layered approach. That means that we must map between objects.

*AutoMapper* is a library that will help make our lives a little easier by performing the mappings for us.

The first mapping we need will map **from** a design model *Customer* class **to** a view model *CustomerBaseViewModel* class. When you write the “create map” statement, ensure that you include the namespace name.

In the *Manager.cs* file (located in the *Controllers* folder), add the mapping.

```
// Configure the AutoMapper components
var config = new MapperConfiguration(cfg =>
{
    // Define the mappings below, for example...
    // cfg.CreateMap<SourceType, DestinationType>();
    // cfg.CreateMap<Employee, EmployeeBase>();

    cfg.CreateMap<Customer, CustomerBaseViewModel>();
});
```

Remember to build your solution regularly to ensure there are no errors in your code.

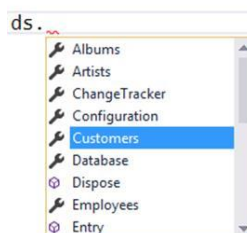
## In the Manager, create a “get all” method

As you have learned, the purpose of the *Manager* class is to “manage” all the data tasks and operations.

The manager has a reference named “**ds**” that references the data store. It also has a reference named “**mapper**” to the instance of *AutoMapper*.

As stated in the comments, ensure that the methods accept and deliver ONLY *view model* objects and collections. The collection return type is almost always *IEnumerable<T>*, this situation is no different.

The object reference knows about all the entity stores:



Customers is a `DbSet<T>` data type. The data context class (`DataContext.cs` in the *Data* folder) defines a `DbSet<T>` property for each entity in the database. `DbSet` and the superclass of `DataContext` (which is `DbContext`) are classes in the [Entity Framework](#). `DbSet` inherits from a number of interfaces, including `IEnumerable<T>`. This will allow the AutoMapper to do its work.

**ds.Customers** is a collection of Customer objects from the data store. We cannot return this collection directly, instead we must transform it into a collection that is defined by a *view model* class.

If you were to do this transformation in code, without AutoMapper, you would typically do something like this:

1. Define a new empty view model collection.
2. Loop through the source collection, maybe with a “for” or “foreach” loop.
3. For each item in the source collection, create a view model object and set each property.

For example:

```
// Manually map objects, source to target
public IEnumerable<CustomerBaseViewModel> CustomerGetAll()
{
    // Define a new empty view model collection
    var results = new List<CustomerBaseViewModel>();

    // Fetch all the objects from the data store
    var allCustomers = ds.Customers;

    // Manually map each source object to a target object
    foreach (var customer in allCustomers)
    {
        // Create and configure a new target object
        var c = new CustomerBaseViewModel();
        c.Address = customer.Address;
        c.City = customer.City;
        c.Company = customer.Company;
        c.Country = customer.Country;
        c.CustomerId = customer.CustomerId;
        c.Email = customer.Email;
        c.Fax = customer.Fax;
        c.FirstName = customer.FirstName;
        c.LastName = customer.LastName;
        c.Phone = customer.Phone;
        c.PostalCode = customer.PostalCode;
        c.State = customer.State;

        // Add the new target object to the results collection
        results.Add(c);
    }

    // Return the results
    return results;
}
```

With AutoMapper, the function above condenses to only a single line.

```
0 references | 0 exceptions
public IEnumerable<CustomerBaseViewModel> CustomerGetAll()
{
    return mapper.Map<IEnumerable<Customer>, IEnumerable<CustomerBaseViewModel>>(ds.Customers);
}
```

Source object type                      Target object type                      Source object

In the Manager, create a “get one” method

If you skimmed the documentation for the [DbSet<T>](#) class, you may have noticed a [Find\(\)](#) method. As stated in the documentation, this method “Finds an entity with the given primary key values ... if no entity is found in the [data] context or the [data] store, then null is returned.”. This function will easily allow us to “get one” object from the store. You will pass the primary key as an argument to the Find function - *remember, the primary key is stored in the CustomerId property*.

See the example function below. First, we attempt to fetch a matching object from the data store. Next, we return the object while guarding against a null result. If the object is not null, AutoMapper is used to transform it into a *view model* object.

```
0 references | 0 exceptions
public CustomerBaseViewModel CustomerGetById(int id)
{
    // Attempt to fetch the object.
    var obj = ds.Customers.Find(id);
    // Return the result (or null if not found).
    return obj == null ? null : mapper.Map<Customer, CustomerBaseViewModel>(obj);
}
```

Pass in the object identifier                      Source object                      Source object type                      Destination object type

Create a controller and add a reference to the manager

First off, create the new controller by right-clicking the Controllers folder and choosing “Add > Controller”. Use the “MVC 5 Controller with read/write actions” template and name the controller “**CustomersController**”.

The controller must have a reference to a manager object. Therefore, at the top of the controller class, declare a private field for the manager, and initialize it during the declaration:

```
// Reference to a manager object
private Manager m = new Manager();
```



## Modify the Index() method to call the manager “get all” method

Above, you wrote the manager “get all” method. We will call that method, which will return a collection of CustomerBase objects, and then pass the collection to the view.

We have a choice to code this task as two statements or as one statement. The choice depends upon whether you need to validate, transform, modify, or manipulate the fetched results before passing them to the view. If you do, you can use a variant of this approach:

```
// Fetch the collection
var c = m.CustomerGetAll();

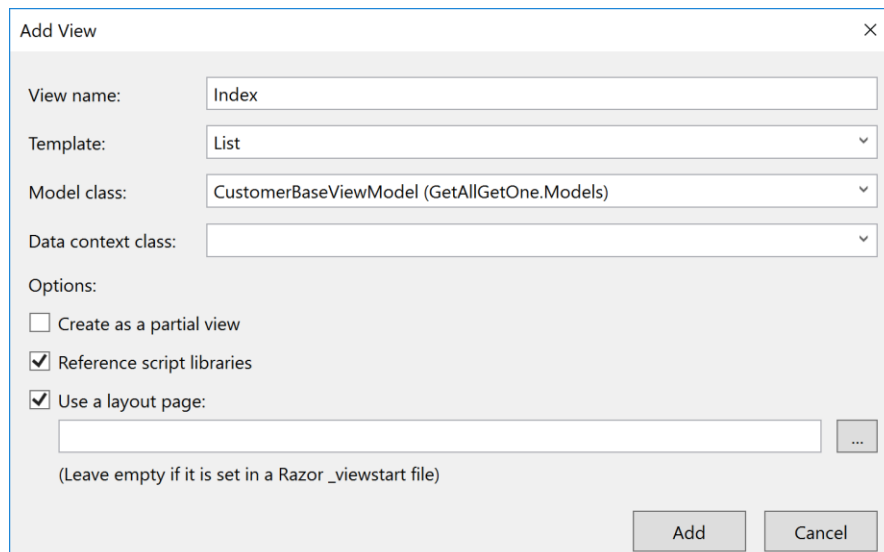
// Pass the collection to the view
return View(c);
```

Alternatively, if the fetched results are ready-to-use, you can simply call the method as the argument to the View() method. With a collection, we do not have to worry about a null result because a collection with zero objects will simply return an empty collection. An empty collection is a valid object. Here’s the code:

```
return View(m.CustomerGetAll());
```

## Create a “get all” view, using the scaffolder

Now, we can create the view. With the cursor anywhere in the Index() method code block, right-click and choose Add View. Complete the dialog as shown below:



The screenshot shows the 'Add View' dialog box with the following configuration:

- View name: Index
- Template: List
- Model class: CustomerBaseViewModel (GetAllGetOne.Models)
- Data context class: (empty)
- Options:
  - ☐ Create as a partial view
  - ☒ Reference script libraries
  - ☒ Use a layout page: (empty text box with a browse button)

At the bottom, there are 'Add' and 'Cancel' buttons. A note at the bottom states: '(Leave empty if it is set in a Razor \_viewstart file)'.

Edit the view to make its appearance and text content nicer. At this point in time, you should run the app (Ctrl+F5). The /Customers/Index URI segment will show the list of customer objects.

## Modify the Details() method, to call the manager “get one” method

Study the signature of the “get one” method. It accepts an *int* argument, presumably the object’s identifier.

There is a problem, that is, we are not guarding against an empty/null identifier value? It is a *best practice* to change the signature of the “get one” method so that the *int* data type can become nullable. In other words, we need to change the **int** to a **nullable int**, written as “**int?**”.

```
public ActionResult Details(int? id)
```

Making the *int* nullable will add a convenience method `GetValueOrDefault()`. Basically, this method works such that if the value is valid, for example 3 or 275, then the value is returned. If the value is NOT valid, for example null or “abc”, then the default value – zero (0) – will be used. This is a safe coding practice.

If the id is null, we will return an `HttpNotFound()` response. Later in the course, you will learn a more elegant way to handle “not found” and other error-like results. If the id is valid, return the appropriate object.

Here’s the complete method body:

```
// GET: Customers/Details/5
public ActionResult Details(int? id)
{
    // Attempt to get the matching object
    var obj = m.CustomerGetById(id.GetValueOrDefault());

    if (obj == null)
        return HttpNotFound();
    else
        return View(obj);
}
```

## Create a “get one” view, using the scaffolder

Create the “get one” view. Complete the dialog as shown below:

Add View

View name: Details

Template: Details

Model class: CustomerBaseViewModel (GetAllGetOne.Models)

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

(Leave empty if it is set in a Razor \_viewstart file)

Add Cancel

Now, going back to the browser, the “Details” link for each of the items on the list of customers will work and will take you to a details view.

Edit the view to make its appearance and text content nicer.

## Customer “Add New” Functionality

We already have a *CustomerBaseViewModel* class containing all of the properties we need to describe a customer object but we can't use it to handle the “add new” use case, why?

As you recently learned, we need a *CustomerAddViewModel* class. It has the properties needed for a new object EXCEPT for the identifier. Why no identifier? By default, and by convention, the database server automatically assigns the identifier.

Note that this design approach says nothing about indexes that you (or your database administrator) may wish to create in a database to support fast lookups. That is an advanced topic not covered in this course.

Create a new *CustomerAddViewModel* class. **Cut, do not copy** all properties EXCEPT the identifier from the *CustomerBaseViewModel* class and paste them into the *CustomerAddViewModel* class.

```
public class CustomerAddViewModel
{
    [Required]
    [StringLength(40)]
    public string FirstName { get; set; }

    [Required]
    [StringLength(20)]
    public string LastName { get; set; }

    [StringLength(80)]
    public string Company { get; set; }

    [StringLength(70)]
    public string Address { get; set; }

    [StringLength(40)]
    public string City { get; set; }

    [StringLength(40)]
    public string State { get; set; }

    [StringLength(40)]
    public string Country { get; set; }

    [StringLength(10)]
    public string PostalCode { get; set; }

    [StringLength(24)]
    public string Phone { get; set; }

    [StringLength(24)]
    public string Fax { get; set; }

    [Required]
    [StringLength(60)]
    public string Email { get; set; }
}
```

We will now modify the *CustomerBaseViewModel* class to [inherit](#) from the *CustomerAddViewModel* class. The syntax is simple and perhaps similar to what you see in other programming languages – simply add a colon and the name of the superclass after the class name.

```
public class CustomerBaseViewModel : CustomerAddViewModel
{
    [Key]
    public int CustomerId { get; set; }
}
```

We will use view model class inheritance often so make sure that you understand the concept and its implementation.

In the Manager, create an “add new” method

Back in the Manager class, we will need to map from the *CustomerAddViewModel* class to the *Customer* class. When you write the “create map” statement ensure that you include the fully-qualified class name.

```
public Manager()
{
    // If necessary, add more constructor code here...

    // Configure the AutoMapper components
    var config = new MapperConfiguration(cfg =>
    {
        // Define the mappings below, for example...
        // cfg.CreateMap<SourceType, DestinationType>();
        // cfg.CreateMap<Employee, EmployeeBase>();

        // Map from Customer design model to CustomerBaseViewModel.
        cfg.CreateMap<Customer, CustomerBaseViewModel>();

        // Map from CustomerAddViewModel to Customer design model.
        cfg.CreateMap<CustomerAddViewModel, Customer>();
    });

    mapper = config.CreateMapper();
}
```

If you skimmed the documentation for the [DbSet<T>](#) class, you may have noticed an [Add\(\)](#) method and a [SaveChanges\(\)](#) method in the [DbContext](#) class.

[Add\(\)](#) will add a new object to the data context. The object is added in-memory not to the database. [SaveChanges\(\)](#) will determine any changes made to the context and will save them to the data store.

Now that we know these two additional functions exist, let’s create an “add new” method. The new method will need an argument of type *CustomerAddViewModel* that can be passed in from the controller. It will also require a return type of *CustomerBaseViewModel* that can be passed back to the controller.

Our first task is to add a new object to the data context *Customers* collection. The [Add\(\)](#) method is expecting an object of type *Customer* therefore we must map from *CustomerAddViewModel* to *Customer*. Next, we will save the changes. Finally, return the object, guarding against a null result, using AutoMapper to transform it into a *view model object*.

Here is a typical “add new” method:

```
public CustomerBaseViewModel CustomerAdd(CustomerAddViewModel newCustomer)
{
    // Attempt to add the new item.
    // Notice how we map the incoming data to the Customer design model class.
    var addedItem = ds.Customers.Add(mapper.Map<CustomerAddViewModel, Customer>(newCustomer));
    ds.SaveChanges();

    // If successful, return the added item (mapped to a view model class).
    return addedItem == null ? null : mapper.Map<Customer, CustomerBaseViewModel>(addedItem);
}
```

### Controller Create() method, for HTTP GET

In the controller modify the first Create() method, if necessary.

In ASP.NET MVC web apps, we ALWAYS use two methods for “add new”, “edit existing”, and “delete item”.

The first method responds to HTTP GET. Its job is to prepare the data that is needed for an HTML Form and pass the data to the view (which appears in the browser).

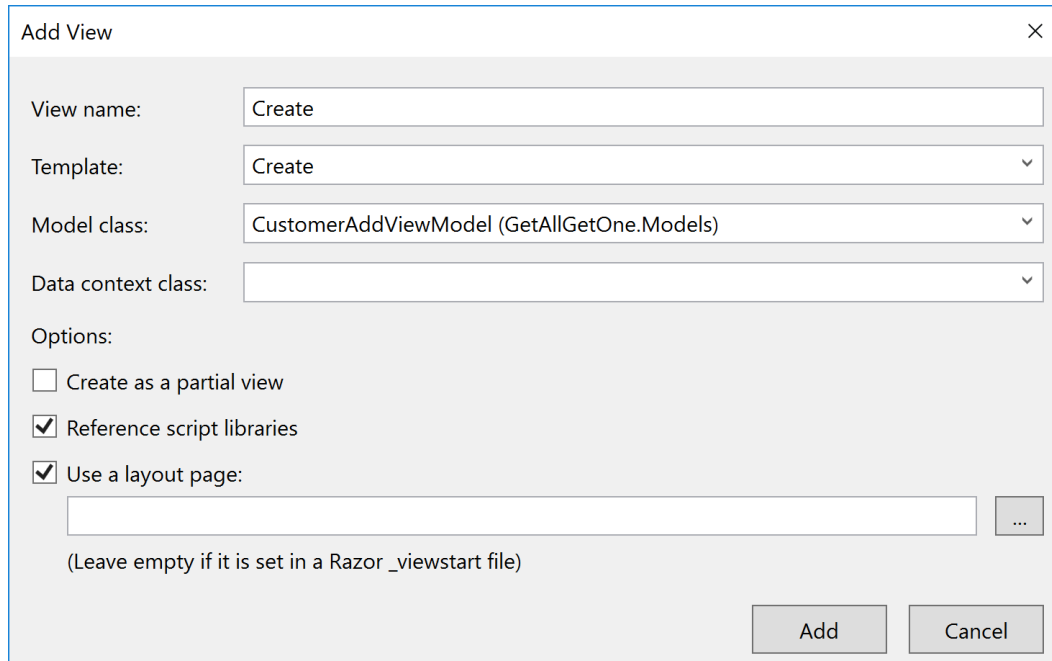
The other method responds to HTTP POST. When the user submits the HTML Form, its job is to validate the incoming data, process it, and handle the result.

The CustomerAddViewModel class is simple, with text-only properties. For this situation, it probably is not necessary to send any data to the HTML Form.

```
// GET: Customers/Create
public ActionResult Create()
{
    // Optionally create and send an object to the view
    return View();
}
```

## Create an “add new” view, using the scaffolder

Create the “get one” view. Complete the dialog as shown below:



If you run the project, the “Create new” link on the customer list page will work. Clicking it will take you to a create view.

## Second Create() method, for HTTP POST

The job of this method is to validate the incoming data, process it, and handle the result.

How does it validate incoming data? An ASP.NET MVC feature named [\*model binding and validation\*](#).

In Assignment 1, your Create() method that handled HTTP POST had an argument of type FormCollection. From now on, we will use **strong typing** and set the argument to the view model type that we are expecting from the posted HTML Form. This is super important so please make sure that you understand what will happen.

```
public ActionResult Create(CustomerAddViewModel newItem)
```

Next, we check whether the incoming object is valid. If not, we redisplay the form with the bad data. The view will show relevant error message(s) to the user. (How? You will learn that later in the course.)

Otherwise, if the object is valid, we can call the manager “add new” method and pass in the object. We expect a fully-configured object to be returned. The returned object will have the unique identifier that was assigned when the object was saved in the data store.

If there was a problem adding or saving, we redisplay the form with the bad data. Otherwise, we handle the result and decide what to do next.

A best practice is that these three kinds of requests – “add new”, “edit existing”, and “delete item” – use a web app pattern named **Post – Redirect – Get** (PRG). The biggest benefit of this pattern is that it prevents problems that may arise from attempts by a user to re-submit an already-processed HTML Form, or refresh a submitted HTML Form, or an attempt at using the browser’s “back” button/functionality on a submitted HTML Form.

How is that pattern implemented? At the end of the method, simply return a `RedirectToAction()` result. This will cause the web app to send a HTTP 302 to the browser with a Location header value that points to the resource you specify.

After “add new”, an acceptable best practice is to redirect to a “details” view, which you already have coded.

Here is the complete code for the method:

```
// POST: Customers/Create
[HttpPost]
public ActionResult Create(CustomerAddViewModel newItem)
{
    // Validate the input
    if (!ModelState.IsValid)
        return View(newItem);

    try
    {
        // Process the input
        var addedItem = m.CustomerAdd(newItem);

        // If the item was not added, return the user to the Create page
        // otherwise redirect them to the Details page.
        if (addedItem == null)
            return View(newItem);
        else
            return RedirectToAction("Details", new { id = addedItem.CustomerId });
    }
    catch
    {
        return View(newItem);
    }
}
```