

# Build a dynamic solution to capture, store and deliver media items

## LECTURE 10.1 HANDS-ON LAB

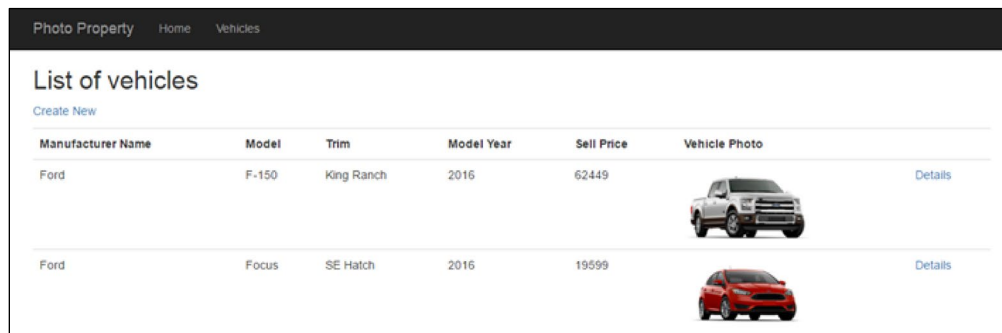
You have already learned how to statically store and deliver media items. (A media item is located in your web app and it can be accessed by a URL). If you have ever used the HTML `<input type=file>` element, then you know how to capture (i.e. upload) a media item; this task requires server-side logic to save the media item in the file system. If you have never done this, you will learn how the capture works and can adapt it.



We will build a dynamic solution to capture, store and deliver media items. Media items will be in the persistent store (i.e. database). Here's a preview of the solution's parts and tasks:

- Entity classes will include properties for the media items' data bytes and media type.
- Capturing a media item is done with a view + view model + controller + manager.
- Delivering a media item is done with a controller + manager.

Open the **PhotoProperty** code example, and study it as you continue reading the next sections.

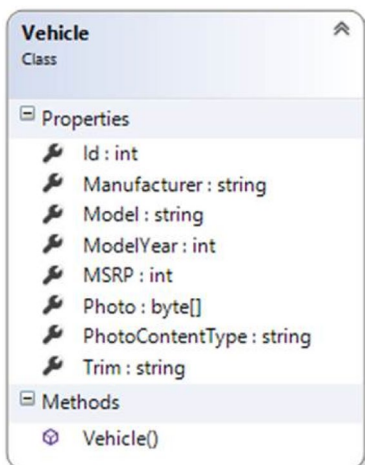
If you want to use the code example, there are some extra photos in the project's App\_Data folder that you can choose/select. If you want accurate model, trim, and price data, it's on the *[ford.ca](http://ford.ca)* web site.



Manufacturer Name	Model	Trim	Model Year	Sell Price	Vehicle Photo	
Ford	F-150	King Ranch	2016	62449		<a href="#">Details</a>
Ford	Focus	SE Hatch	2016	19599		<a href="#">Details</a>

## Entity class design

We return to the familiar vehicle / manufacturer problem domain. We will only have a **Vehicle** class. As noted above, it has properties for the media item's data and media type:



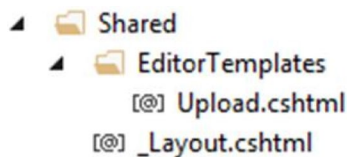
Capturing a media item is done with a view + view model + controller + manager

**VehicleAddFormViewModel Class:** This class is used to build an object for the HTML Form. It will have the expected properties, including a **PhotoUpload** property (of type string) that will use a **DataType** data annotation to identify it as a file upload property.

```
[Required]
[Display(Name = "Vehicle Photo")]
[DataType(DataType.Upload)]
public string PhotoUpload { get; set; }
```

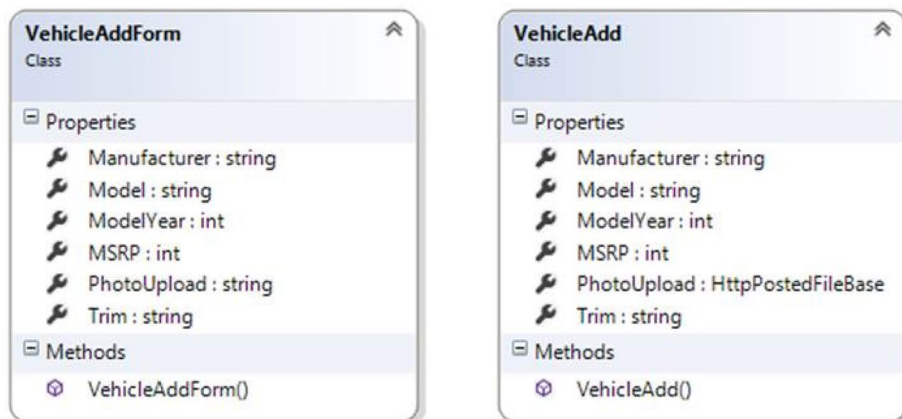
Normally, when you ask the scaffolder to build a “create” view for a string property that uses **DataType.Upload**, the scaffolder will NOT generate the code in the view. You would have to add it yourself.

To save a bit of work, the code example includes an editor template, which is a feature of ASP.NET MVC. In Solution Explorer, look in the “Views > Shared” folder and you will see a partial view, *Upload.cshtml*. Its code will be automatically used by the scaffolder to generate the code for a **DataType.Upload** string property.



**VehicleAddViewModel Class:** It describes the data submitted by the user. It also has a **PhotoUpload** property but its type is **HttpPostedFileBase**. As noted in this [MSDN document](#), it is “the base class ... that provides access to individual files that have been uploaded by a client”.

Compare these two classes, side-by-side, you will see the only difference is the *Type* of the **PhotoUpload** property. Before continuing, check that the AutoMapper “create map” statements have been defined.



**The Controller:** The good news is that the configuration of the two “add new” methods (GET and POST) is similar to almost every other “add new” use case.

**The View:** Use the scaffolder to generate the **Create** view then edit it to clean up its title, etc. There is an extremely important edit that you must make to the view since the scaffolder will not do it for you. As you have seen in all scaffolded Create views, the HTML Form is defined by this code:

```
@using (Html.BeginForm())
```

That specific constructor will NOT configure the form to allow file uploads. We must write our own constructor. Use the following, which will work for almost all scenarios that you'll code:

```
@using (Html.BeginForm(null, null, FormMethod.Post, new { enctype = "multipart/form-data" }))
```

Photo Property Home Vehicles

### Create vehicle

Complete the form, and click the Create button

Model

Trim

Model Year

Sell Price

Manufacturer Name

Vehicle Photo  No file chosen

[Back to List](#)

**Manager Class:** Its “add new” method will look similar to all others that you’ve coded but it will have more statements to handle the uploaded media item.

Notice that the view model class properties for **PhotoUpload** have the [Required] attribute. Therefore, if the code in the manager method is running, you can be assured that it has a **PhotoUpload** object and therefore you don’t have to test for null etc.

Here’s the code for a typical “add new” manager method that handles an uploaded media item:

```
public VehicleBase VehicleAdd(VehicleAdd newItem)
{
    // Attempt to add the new item
    var addedItem = ds.Vehicles.Add(Mapper.Map<VehicleAdd, Vehicle>
(newItem));

    // Handle the uploaded photo...

    // First, extract the bytes from the HttpPostedFile object
    byte[] photoBytes = new byte[newItem.PhotoUpload.ContentLength];
    newItem.PhotoUpload.InputStream.Read(photoBytes, 0,
newItem.PhotoUpload.ContentLength);

    // Then, configure the new object's properties
    addedItem.Photo = photoBytes;
    addedItem.PhotoContentType = newItem.PhotoUpload.ContentType;

    ds.SaveChanges();

    return (addedItem == null) ? null : Mapper.Map<Vehicle,
VehicleBase>(addedItem);
}
```

A few notes about the Manager class “get...” methods and the “...Base” class: The manager methods look the same as other methods that you have written. As noted earlier, the “...Base” class does NOT (and must not) include properties for the media item. Therefore, we are not leaking – unintentionally or otherwise – media item data to the controllers and views. We continue to work with relatively small-sized objects in memory.

## Delivering a media item is done with a controller + manager

At this point in time, we have a way to save a media item into the persistent store (i.e. database). Now let's see how we can get it out of the database and deliver it to a user.

In our dynamic approach, we will use a special-purpose controller to deliver a media item. It will use attribute routing to shape the URL segment. The result will be that the media item is available to the user at a specific URL.

Above, you learned that the "...Base" class does not include media item properties but now we will need them. We can create a new view model class, perhaps named "...Media" or "...Photo". It will be a really simple class with few properties:



In the Manager class, the **VehiclePhotoGetById()** method looks like any other "get one" method, and it returns a **VehiclePhoto** object. (There is an AutoMapper "create map" defined.)

## Designing the special-purpose media item delivery controller

In the code example, a **PhotoController** was created. Its singular purpose will be to host a "get one" method that accepts an identifier for a media item and then deliver its to the user. It will use attribute routing to shape the URL, so that the web app programmer and the user need only use a very simple URL format:

/photo/123

The code will look as follows. Notice that it is NOT returning a view ActionResult. Instead, it is returning a file ActionResult. The File() constructor arguments will cause the request-handling pipeline to construct a correct response.

```
// GET: Photo/5
[Route("photo/{id}")]
public ActionResult Details(int? id)
{
    // Attempt to get the matching object
    var o = m.VehiclePhotoGetById(id.GetValueOrDefault());

    if (o == null)
    {
        return HttpNotFound();
    }
    else
    {
        // Return a file content result
        // Set the Content-Type header, and return the photo bytes
        return File(o.Photo, o.PhotoContentType);
    }
}
```

## Vehicle details

**Model** Focus  
**Trim** SE Hatch  
**Model Year** 2016  
**Sell Price** 19599  
**Manufacturer Name** Ford  
**Vehicle Photo**



[Back to List](#)