

Build **another** dynamic solution to capture, store and deliver media items

LECTURE 10.2 HANDS-ON LAB

Recently, you began to learn how to handle non-text media types in an ASP.NET MVC web app. The sample solution that we worked with had these characteristics:

- A design model class was configured with properties to hold the digital content of a single media item, in addition to the properties that held text and numbers.
- The web app allowed or enabled a browser user to “upload” a media item, when creating a new object.
- The digital content of the media item (bytes, string for content type) was stored in the web app’s persistent store (i.e. the database).
- Finally, the web app delivered the media item, in its native format, upon request.

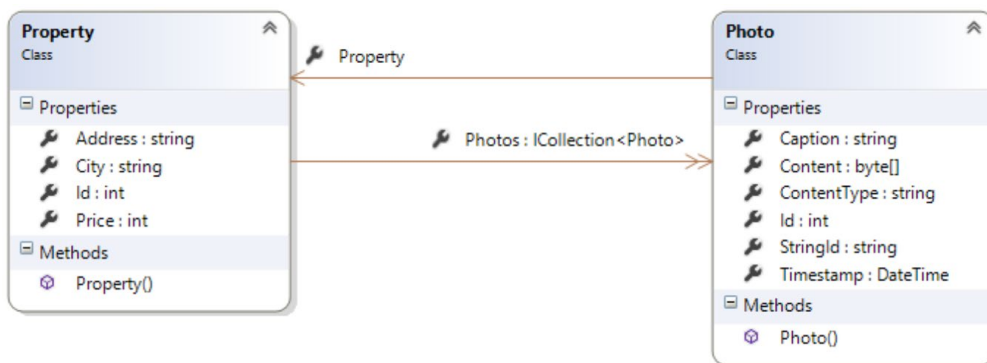
While this kind of sample solution was suitable for an object that could also be represented by its media item, it’s not suitable for all situations. We will learn how to build on the learned concepts and techniques, to build a larger solution.

The New Problem

In many situations, an object is associated with a collection of media items. For example, a product could be associated with a collection of photos, each showing a different part of the product.

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

The most important idea in this code example is a dedicated design model class for the media item. It has properties to hold the digital content, as well as other identifying and descriptive properties. Study the image below and the code for the Photo design model class in the code example.



Identifier comments

In the other scenario – where a class included media item properties in its definition – we could use the object identifier to work with the object or its media item. The representation of the object could have been the HTML or maybe JSON data, or it could have been the media item. As a result, there was no ambiguity with the identifier.

In this new scenario, we want to discourage the use of the **int** identifier that’s generated by the database/storage engine. Instead, we should create an identifier. We really do not want to use an identifier that’s influenced by the original name or content of the media item. For example, before uploading, a photo could be saved as “fuzzy-kitten.jpg” on someone’s computer. In this new scenario, we do not want to use “fuzzy-kitten” or any value that the user provides.

We will generate our own identifier, as the media item is being prepared for storing/saving. We start with the known principle that a GUID – globally unique identifier – is, or will be, as unique as we need it to be.

A minor problem is that the standard format of a GUID is 36 characters in length (hex digits and dash separators). That’s a bit too long. Mads Kristensen [thought of a way](#) to capture the uniqueness of a GUID in a 16-character string (*looks like it is now down to 10 characters*). We’ll use that. Study the constructor in the design model class to see its code.

The benefit is that we can now create a controller to deliver media items and use the string identifier as part of the URL (instead of the int “Id” value). For example: **<http://www.example.com/photo/3c4ebc5f5f2c4edc>**

For an object, rendering its collection of media items

The “**PhotoEntity**” code example covers the residential real estate sales problem domain. It enables a user to create an object that represents a for-sale residential property. The user can add photos to this existing object.

The design and coding approach for upload is exactly the same as the one we have used before: Add an associated object, for a known existing object. For delivery (i.e. photo viewing), we adapt the object-with-collection delivery approach.

In the code example, start by looking at the list of for-sale residential properties. It has a new link.

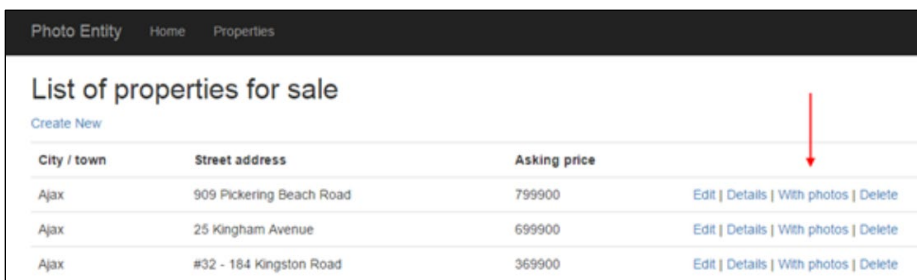


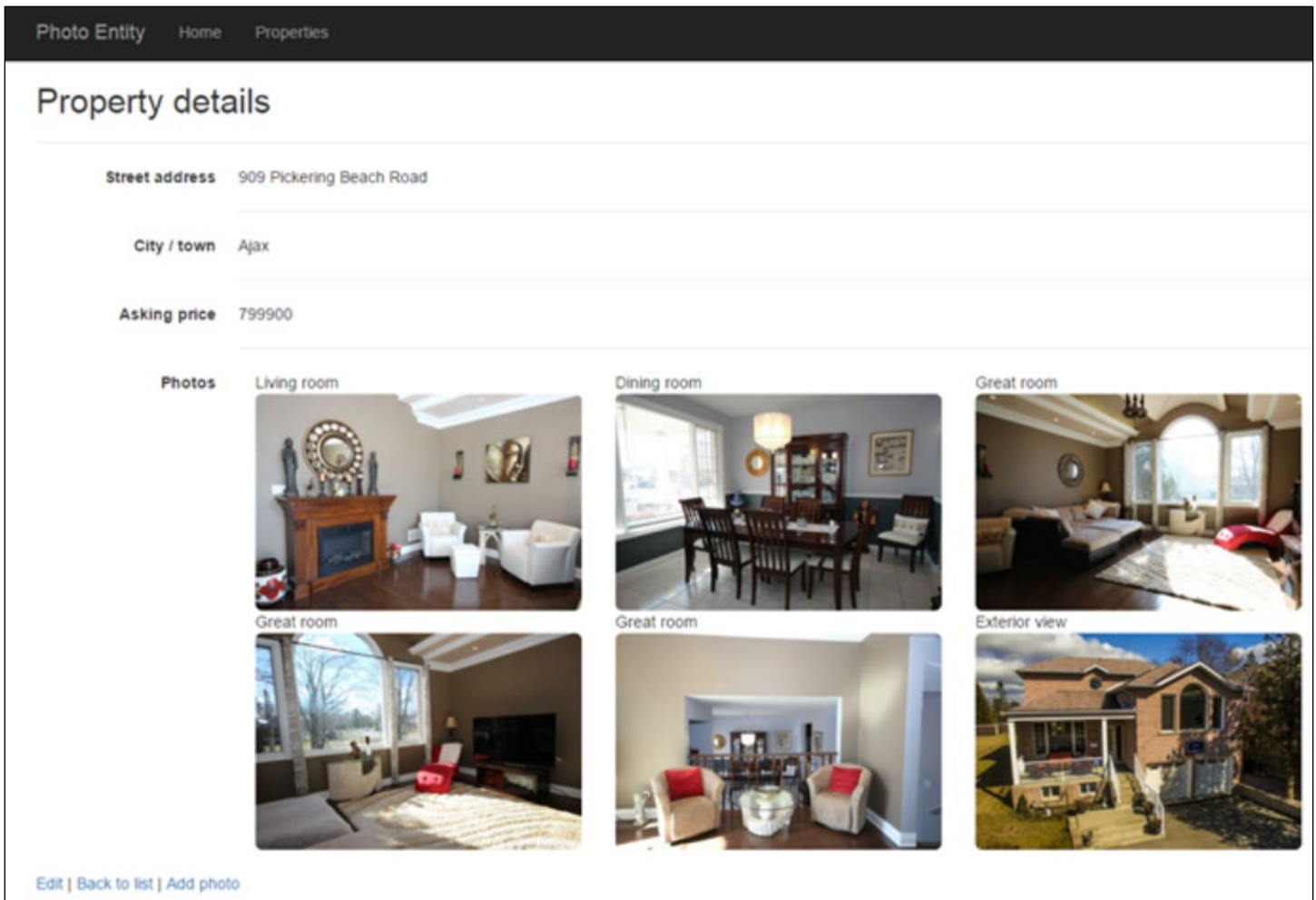
Photo Entity Home Properties			
List of properties for sale			
Create New			
City / town	Street address	Asking price	
Ajax	909 Pickering Beach Road	799900	Edit Details With photos Delete
Ajax	25 Kingham Avenue	699900	Edit Details With photos Delete
Ajax	#32 - 184 Kingston Road	369900	Edit Details With photos Delete

Alternatively, from the standard “details” page. It too has a new link.



Photo Entity Home Properties	
Property details	
Street address	909 Pickering Beach Road
City / town	Ajax
Asking price	799900
Edit Details with photos Back to list Add photo	

When you follow the link, you will see a “details” view that renders the photos.



To make this work, we create a PhotoBaseViewModel class:

```
// FYI - data annotations were removed to help clarify the concept
public class PhotoBase
{
    public int Id { get; set; }
    public DateTime Timestamp { get; set; }
    public string StringId { get; set; }
    public string Caption { get; set; }
}
```

A **Photo** object (from the persistent store) maps enough properties to the **PhotoBaseViewModel** object to allow us to construct a URL to the photo in our view code.

The controller's "get one" method calls a Manager class method that fetches/includes the object's photos. Here's what the view model class looks like for the object-with-a-photo-collection.

```
public class PropertyWithPhotoStringIds : PropertyBase
{
    public PropertyWithPhotoStringIds()
    {
        Photos = new List<PhotoBase>();
    }

    public IEnumerable<PhotoBase> Photos { get; set; }
}
```

Here's what the relevant view model classes look like:



In the view, we loop through the object's collection of photos, and render an HTML element.

```
<dd>
  @if (Model.Photos.Count() == 0)
  {
    <span>(none)</span>
  }
  else
  {
    foreach (var item in Model.Photos)
    {
      <div class="col-md-4 col-sm-5 col-xs-10">
        <span>@item.Caption</span><br>
        
      </div>
    }
  }
<hr>
</dd>
```

In the dedicated photo delivery controller, the “get one” photo method uses a string identifier, which is a bit different from the other code example and scenario which used the containing object's **int** identifier. Other than that, it works the same as before.

Bonus feature – allow download-and-save of a media item

The dedicated photo delivery controller includes another “get one” photo method, with a modified attribute routing string (**photo/{stringId}/download**). Its purpose is to enable the download-and-save workflow. Sometimes you will want to do this but it may not always be appropriate to render a media item in a browser. Try using the code example to fetch-and-save an image.

Study the code in the method, it is well-commented to explain what it does. Notice the extra code that generates a file extension, which is needed for the “save” task. Is there another way to do this?

Probably, by saving the file extension in the “add/upload photo task”, which implies that the design model class would need changes.

Reminder – making changes to the design of the persistent store

After an app has been running for a few hours/days/weeks, how do you handle a situation where you need to add another entity class to a design model?

Or, add a property to an existing design model class?

Or, change a property’s configuration or data annotations?

This can be done if you activate a feature named (Code First) Migrations. Its biggest benefit is that it will attempt to keep your existing data after minor changes to the design model and database.

Another benefit is that the feature enables us to publish our web app to a public host.