**Agenda for the Monday class**

Here’s the agenda for the Monday class:

* Highlights of the design of a media type formatter
* Media item delivery, and content negotiation
* Work on the assignment

**Design and implement a media type formatter**

Above, we stated that we can design and implement a media type formatter. How?

Here’s the general approach to handling images, or *any* byte-oriented content:

1. Create (or add) and register a custom media type formatter
2. Configure the design model class with properties to hold the media item
3. Configure resource model classes with media-related properties
4. Add manager methods to handle media-related acceptance (and delivery)
5. Configure controller methods

Continue reading below, to learn the details. Study the code example while you do that.

**Create (or add) and register a custom media type formatter**

We need a formatter that will handle byte-oriented content. That will enable us to handle these media types:

* images/photos (png(Portable Network Graphics), jpg(Joint Photographic Experts Group - JPG is a file extension for a lossy graphics file), etc.)
* audio (m4a(MPEG 4 Audio - a file extension for an audio file encoded with advanced audio coding (AAC) which is a lossy compression), mp3(Metroid Prime 3), etc.)
* video (mp4(MPEG-4 Part 14 - digital multimedia container format), wmv (Windows Media Video), mov, etc.)
* documents (pdf, docx, xlsx, zip, etc.)

Create a class that inherits from BufferedMediaTypeFormatter. In the code example, study the ByteFormatter class (it’s found in the project’s ServiceLayer folder).

All formatters have a similar structure:

* a *constructor* configures the media types that it will handle
* a *reader* method transforms incoming content from the HTTP request entity body
* a *writer* method transforms outgoing content to the HTTP response entity body

Note: This formatter design works well for small- to medium-sized media items. If your media item is less than a few hundred kilobytes, then you can use this design, which is *synchronous* and therefore *blocking*.

For larger media items, use the *asynchronous* approach, where the base class is simply MediaTypeFormatter. A future code example will show you how to do this.

The formatter must be registered and initialized when the app starts. Look at the WebApiConfig class, in its Register method, for the statement that does this work.

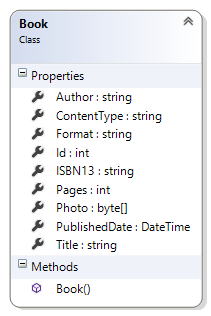
At this point in time, the web service can handle these media types in its requests and responses, and automatically transform/convert between them and C# objects:

1. JSON
2. XML
3. byte-oriented media items

**Configure the design model class with properties to hold the media item**

You need two properties:

1. A byte array (byte[]) for the photo (named “PhotoMedia” or something similar), and
2. A string for the internet media type (named “PhotoContentType”).

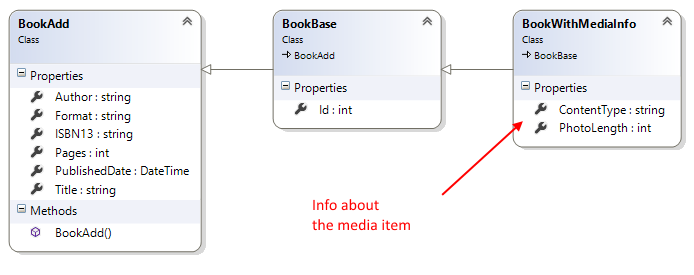


**Configure resource model classes with media-related properties**

Remember the rule above: In a request or response *data package*, do **NOT** package a media item inside another container.

However, during processing, while doing work between the controller, to/from the manager, to/from the data store, it is natural and obvious to consider the media item to be a property of an entity object.

Resource models can and should acknowledge the presence of the media item. Study the diagram below, which shows the code example’s resource model classes.



The *BookWithMediaInfo* class includes metadata about the media item. The controller and manager can use the metadata to make processing decisions. The requestor can also do that.

**Add manager methods to handle media-related acceptance (and delivery)**

Today’s code example *MediaUpload* focuses on the acceptance (upload) of a media item.

Above, you learned that an entity object must exist before its media item can be configured. This situation is similar to any PUT request that modifies properties in an existing entity object.

Therefore, the manager method’s signature includes parameters for the object’s identifier, in addition to the media item properties.

**Configure controller methods**

The code example has a *BookPhoto()* method that handles the request, which includes:

1. The request URI, with the entity object identifier
2. A request header (Content-Type), which declares the media type of the entity body
3. An entity body, which holds a byte stream

Study the method parameters:

* an identifier for the object
* a byte array for the media item

We can easily understand the identifier. For example, we will be working with object with Id = 234.

What about the request header? It is available in the controller’s Request property.

What about the byte array? How does it get materialized from the request’s entity body?

The *media type formatter* does this work. Automatically.

**Studying the code example**

Here are a few other design features of the *MediaUpload* code example.

Manager:

The “BookGetAll”, “BookGetById” and “BookAdd” methods return metadata for the media item. They do NOT return the media item bytes.

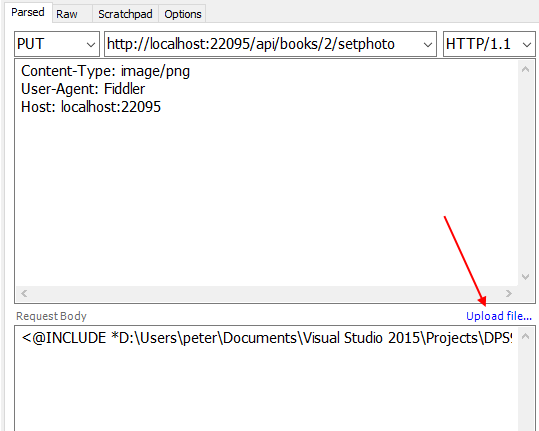
Controller:

As noted above, the “get” and “add new” methods return metadata for the media item. You can inspect the property values to determine whether an “add new” request was successful (and resulted in the acceptance/upload of a media item).

When testing the “BookPhoto” command method in Fiddler, use this technique:

1. Set the method to PUT, and set the appropriate URI (/api/books/{id}/setphoto)
2. For the request body, click the “Upload file…” link, and browse to the media item you wish to use.

This is what it should look like in the Fiddler “composer” tab, after you have selected the media item:



**Delivering a media item in a response**

There are two techniques for delivering a media item in a response:

1. Determined by the web service (and you, the programmer)
2. Negotiated between the requestor and the web service

**Predetermined**

There’s a new *MediaUploadAndDeliver* code example today. Open and study it as you go through the content below.

It has a GetPhoto() method, which returns a media item.

The request includes a URI, and no other configuration. In other words, the request does not need an Accept header configured, because the method simply returns the media item as is.

**Negotiated**

The other “get one” method, Get(), uses *content negotiation*.

The request includes a URI. It also includes an Accept header.

The method will attempt to fulfill this request. If it cannot, it will process the request in a predetermined manner (which is decided by the programmer).

**Content negotiation**

Content negotiation is [described formally in RFC 7231](https://tools.ietf.org/html/rfc7231#section-3.4). Read that section now.

It is also described in [this Wikipedia article](https://en.wikipedia.org/wiki/Content_negotiation).

In summary:

The requestor and the server must *negotiate* the media type of the response.

If the requestor has no preference (i.e. the request does NOT have an Accept request header), the server will deliver its preferred media type.

If the requestor includes an Accept request header, the server will attempt to deliver the resource as the requested media type. You, as the web service programmer, have complete control over this process.

What if the requested media type is not available?

What do you deliver as a response?

[This RFC 7231 section](https://tools.ietf.org/html/rfc7231#section-6.5.4) clearly states that you must return HTTP 404, because “the origin server did not find a current representation for the target resource”.

**Implementing content negotiation**

The *MediaUploadAndDelivery* code example is based on the previously-studied *MediaUpload*code example. As its name suggests, it adds “delivery” of media items as a feature. The following additions/changes were made:

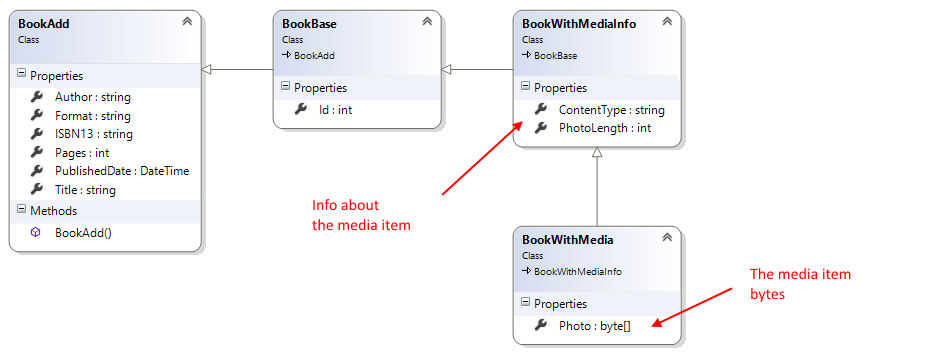
1. Resource model class
2. Manager method return types
3. Controller methods

Each is described below in more detail.

**Resource model class**

A new *BookWithMedia* resource model class was created. It includes the bytes for the media item.

It should be used only in scenarios where you are working with one entity object. In other words, do not do a “get all” and deliver a *BookWithMedia* collection to another place in your code. Why? Think about it: If you have thousands of entity objects, and each object includes a large media item, then the memory and processing load on the app server will be immense.



**Manager method return types**

The “get one” method DOES return the media item bytes, and the metadata.

A controller method will determine whether the media item bytes are returned in the response.

**Controller methods**

There are two methods that service the “get one” use case:

1. GetPhoto(int? id), URI is /api/books/{id}/photo
2. Get(int? id), URI is /api/books/{id}

Remember the rule: In a request or response, do NOT package a media item inside another container.

1. GetPhoto(int? id), URI is /api/books/{id}/photo

This method implements the *predetermined* technique.

Notice the custom route. When using this route, the requestor really is intending to get the media item in the response.

It is not necessary for the requestor to include an Accept header in the request. The method will simply return the media item in the response, and set the Content-Type header accordingly. This approach is similar to the one seen in web app (that uses a browser as a user interface), where the method returns a File() result.

2. Get(int? id), URI is /api/books/{id}

This method implements the *negotiated* technique.

Notice the logic flow:

* It looks for a suitable Accept header value
  + If available, it looks for a media item
    - If available, it delivers the media item; the media type formatter transforms the byte array to a byte stream
    - If not available, it responds with HTTP 404
  + If not available, it handles this as a “normal” request, and responds with a JSON result

The important point is that the method’s return value is a byte array – the actual media item. The custom *media type formatter* transforms the byte array to a byte stream in the response.

**Extra topic: Object with a collection of media items**

This new section covers the second scenario in the previous class/session notes section titled “Media Handling Scenarios”, where an object could have a collection of zero-or-more media items.

Here’s how to get started with its implementation.

**Create a design model class to hold the media item**

First, create a design model class to hold the media item.

It obviously needs a byte array for the media, and a string for media type.

Also, add any other metadata properties that would be useful in your app. What will be useful? Think about the reason for your app, and for existence of the media items. Think about the use cases. Use the results to answer the question.

**Configure the design model classes to have an association**

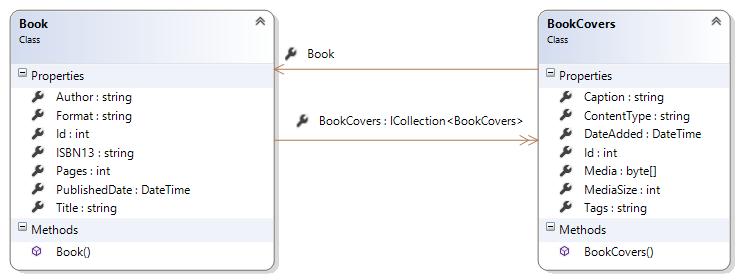
In the “*MediaUpload*” and “*MediaUploadAndDeliver*” code examples, the entity class was Book.

Modify the design of Book, so that it does NOT have media properties. Instead, it will have a collection property that points to the media item class.

Here’s an example of what this design could look like:

**Configure the manager and controller logic**

The “add media item” approach is similar in concept:



First, create the principal item, the Book.

Then, knowing the Book identifier, you can create (POST) a new media item to the BookCovers collection, associating it with the principal Book item.

Finally, knowing the BookCover identifier, you can set its photo property (PUT), using the technique you learned recently. Obviously, in the manager method, it will end up doing an “add new” BookCover task.

**Summary**

Today, we introduced content negotiation, and the concepts and techniques for handling internet media types other than JSON and XML.

You learned about a *media type formatter*, which gets added to the request-handling pipeline.

We also covered the tasks that need to be completed to support these media items in your web service.

**Work on the assignment**

Continue working on the [next assignment](https://petermcintyre.com/dps907/graded-work/assign3/).