**DPS907 notes – week 4 – Sep 28 and Oct 1**

Internet media types in web services.

 <<<<<<<<<<<<

SOAP and REST can't be compared directly, since the first is a protocol (or at least tries to be) and the second is an architectural style. This is probably one of the sources of confusion around it, since people tend to call REST any HTTP API that isn't SOAP.

the main difference between SOAP and REST is the degree of coupling between client and server implementations. A SOAP client works like a custom desktop application, tightly coupled to the server. There's a rigid contract between client and server, and everything is expected to break if either side changes anything. You need constant updates following any change, but it's easier to ascertain if the contract is being followed.

A REST client is more like a browser. It's a generic client that knows how to use a protocol and standardized methods, and an application must fit inside that. You don't violate the protocol standards by creating extra methods, you leverage on the standard methods and create the actions with them on your media type. If done right, there's less coupling, and changes can be dealt with more gracefully. A client is supposed to enter a REST service with zero knowledge of the API, except for the entry point and the media type. In SOAP, the client needs previous knowledge on everything it will be using, or it won't even begin the interaction. Additionally, a REST client can be extended by code-on-demand supplied by the server itself, the classical example being JavaScript code used to drive the interaction with another service on the client-side.

I think these are the crucial points to understand what REST is about, and how it differs from SOAP:

* REST is protocol independent. It's not coupled to HTTP. Pretty much like you can follow an ftp link on a website, a REST application can use any protocol for which there is a standardized URI scheme.
* REST is not a mapping of CRUD to HTTP methods. Read [this](https://stackoverflow.com/questions/19843480/s3-rest-api-and-post-method/19844272#19844272) answer for a detailed explanation on that.
* REST is as standardized as the parts you're using. Security and authentication in HTTP are standardized, so that's what you use when doing REST over HTTP.
* REST is not REST without [hypermedia](https://stackoverflow.com/a/29586455/1202421) and [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS). This means that a client only knows the entry point URI and the resources are supposed to return links the client should follow.

Those fancy documentation generators that give URI patterns for everything you can do in a REST API miss the point completely. They are not only documenting something that's supposed to be following the standard, but when you do that, you're coupling the client to one particular moment in the evolution of the API, and any changes on the API have to be documented and applied, or it will break.

* REST is the architectural style of the web itself. When you enter Stack Overflow, you know what a User, a Question and an Answer are, you know the media types, and the website provides you with the links to them. A REST API must do the same. If we designed the web the way people think REST should be done, instead of having a home page with links to Questions and Answers, we'd have a static documentation explaining that in order to view a question, you have to take the URI stackoverflow.com/questions/<id>, replace id with the Question.id and paste that on your browser. That's nonsense, but that's what many people think REST is.

This last point can't be emphasized enough. If your clients are building URIs from templates in documentation and not getting links in the resource representations, that's not REST. Roy Fielding, the author of REST, made it clear on this blog post: [REST APIs must be hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

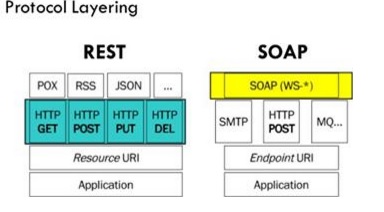
<<<<<

* A REST API should not be dependent on any single communication protocol, though its successful mapping to a given protocol may be dependent on the availability of metadata, choice of methods, etc. In general, any protocol element that uses a URI for identification must allow any URI scheme to be used for the sake of that identification. *[Failure here implies that identification is not separated from interaction.]*
* A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols, such as HTTP’s PATCH method or Link header field. Workarounds for broken implementations (such as those browsers stupid enough to believe that HTML defines HTTP’s method set) should be defined separately, or at least in appendices, with an expectation that the workaround will eventually be obsolete. *[Failure here implies that the resource interfaces are object-specific, not generic.]*
* A REST API should spend almost all its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). *[Failure here implies that out-of-band information is driving interaction instead of hypertext.]*
* A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations. *[Failure here implies that clients are assuming a resource structure due to out-of band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC’s functional coupling].*
* A REST API should never have “typed” resources that are significant to the client. Specification authors may use resource types for describing server implementation behind the interface, but those types must be irrelevant and invisible to the client. The only types that are significant to a client are the current representation’s media type and standardized relation names. *[ditto]*

>>>>>>

With the above in mind, you'll realize that while REST might not be restricted to XML, to do it correctly with any other format you'll have to design and standardize some format for your links. Hyperlinks are standard in XML, but not in JSON. There are draft standards for JSON, like [HAL](http://stateless.co/hal_specification.html).

Finally, REST isn't for everyone, and a proof of that is how most people solve their problems very well with the HTTP APIs they mistakenly called REST and never venture beyond that. REST is hard to do sometimes, especially in the beginning, but it pays over time with easier evolution on the server side, and client's resilience to changes. If you need something done quickly and easily, don't bother about getting REST right. It's probably not what you're looking for. If you need something that will have to stay online for years or even decades, then REST is for you.



>>>>>>>

**Agenda for the Friday class**

Here’s the agenda for the Friday class:

* Test 1
* Transition your knowledge about media handling to the web service context
* Rules (guidance)
* Media type formatter class and code

**Test today**

Test 1 is today, at the beginning of the timeslot.

As discussed in class a few times, it will cover the [week 3](https://petermcintyre.com/dps907/notes/week03/) content and the week 4 content.

Review the relevant info about tests on the [week 2](https://petermcintyre.com/dps907/notes/week02/) notes page and on the [graded work](https://petermcintyre.com/dps907/graded-work/) page.

The test will begin at the start of the class timeslot, at 3:20pm, and its duration will be about 40 minutes. It is worth 12%, which means that you will answer any 12 questions, as each is worth one mark.

After the test, we will cover the highlights of the week’s new topics.

**Code examples**

In the GitHub [code example repository for this week](https://github.com/peteratseneca/dps907fall2018/tree/master/Week_04), you will see several interesting assets.

The *MediaUpload* code example will be studied today, and discussed in the notes below.

Soon, another code example, *MediaUploadAndDeliver*, will be posted, and we’ll likely discuss that in the next class/session.

The *ByteFormatter* folder has a fully-working and complete *media type formatter*, built with the instructions that are in the notes below.

**Reminder – use the Visual Studio Task List…**

Make sure that your Visual Studio app is configured to use the “ATTENTION” comment token.

Then use Visual Studio’s “Task List” to help you study the design and coding features of the code example.

**Transition your web apps media-handling knowledge**

All students have experience with media handling in web apps. In this section, you can begin to transition your knowledge to the web services context.

Refresh your memory about media handling in web apps…

Read and study the professor’s class notes and code examples on this topic, recently covered in a web apps course:  
BTI420 [Class Notes – March 22, 2017](https://petermcintyre.com/bti420/notes/mar22/)  – working with internet media types, simple  
BTI420 [Class Notes – March 24, 2017](https://petermcintyre.com/bti420/notes/mar24-mar27/) – continued, more complex scenario

It will take about an hour for you to go through that content, and its code examples. Do it now. Before continuing.

Welcome back.

In a web app, you learned how to *accept* (i.e. “upload”) a media item, and how to *deliver* a media item. You probably coded these kinds of solutions:

Accept (upload)

* HTML Form
* Includes an “input type=file” element
* The form may also have included other input elements

Deliver

* Specialized controller, or specialized methods in an existing controller
* Returns a File() ActionResult object

**The web service solution**

Accept (upload)

Well, a web service obviously does not have an HTML Form. Therefore, accepting a media item must be done as an atomic and specialized task. The requestor must send only the media item, and its [media type](https://en.wikipedia.org/wiki/Media_type) (as a content type header), in the request.

If the media item is to be “linked” to an entity object, then that object *must* exist, before adding/uploading the media item.

In other words, if you are coding a solution that enables a requestor to, for example, create a new “product” object that has a photo as one of its properties, the solution requires the requestor to create *two separate requests*. First, create the “product” object by supplying data that can be expressed in the “application/json” media type. Then, using the new object’s identifier, create another request to modify the object, and send along the media item as an “image/png” media type.

Deliver

This part of the solution will be similar to the web app solution. The only difference is that the method return type will be different, because it can take advantage of some functionality that you can build into the web service.

**Media handling scenarios**

There are at least three scenarios for handling media:

Object with one media item  
An entity that can also be *represented* with a media item.  
For example, a Seneca “student” object has a single photo item.  
Or, a “song” object has a single audio (e.g. m4a, mp3) item.

Today’s code example covers this object-with-one-media-item scenario: *MediaUpload*.

Object with many media items  
An entity that may have zero or more associated media items.  
This scenario requires a separate “media” entity, associated to its parent entity.  
For example, a “product” object has a collection of product photo items.

Media driven app  
In this scenario, the media drives the app’s purpose and design.  
The “media” entity is central to the design model, and will be associated (by definition, or logically) with other entities.  
For example, a photo app, like Instagram.  
Or, a video app, like YouTube.  
Or, a music app, like Apple Music.  
Or, a document (file) app, like DropBox.

**Rules**

There are at least two important rules that you must follow when working with media items:

Rule 1:  
In a request or response, do **NOT** package a media item inside another container.

Never.

Accept or deliver the media item as-is, even if it means that the requestor must create another request to fetch the item.

Rule 2:  
When storing a media item, its media type name *MUST* also be stored.

Therefore, two properties are required, in a design model class, to store a media item:  
1. A byte array (in C# + Entity Framework + SQL Server), maybe named “PhotoContent” or “PhotoMedia” (assuming that we are storing a photo), and  
2. A string for the type name, maybe named “PhotoContentType”  
Yes, you can add other metadata properties (e.g. location, caption, tags, etc.) for the media item, if you wish.

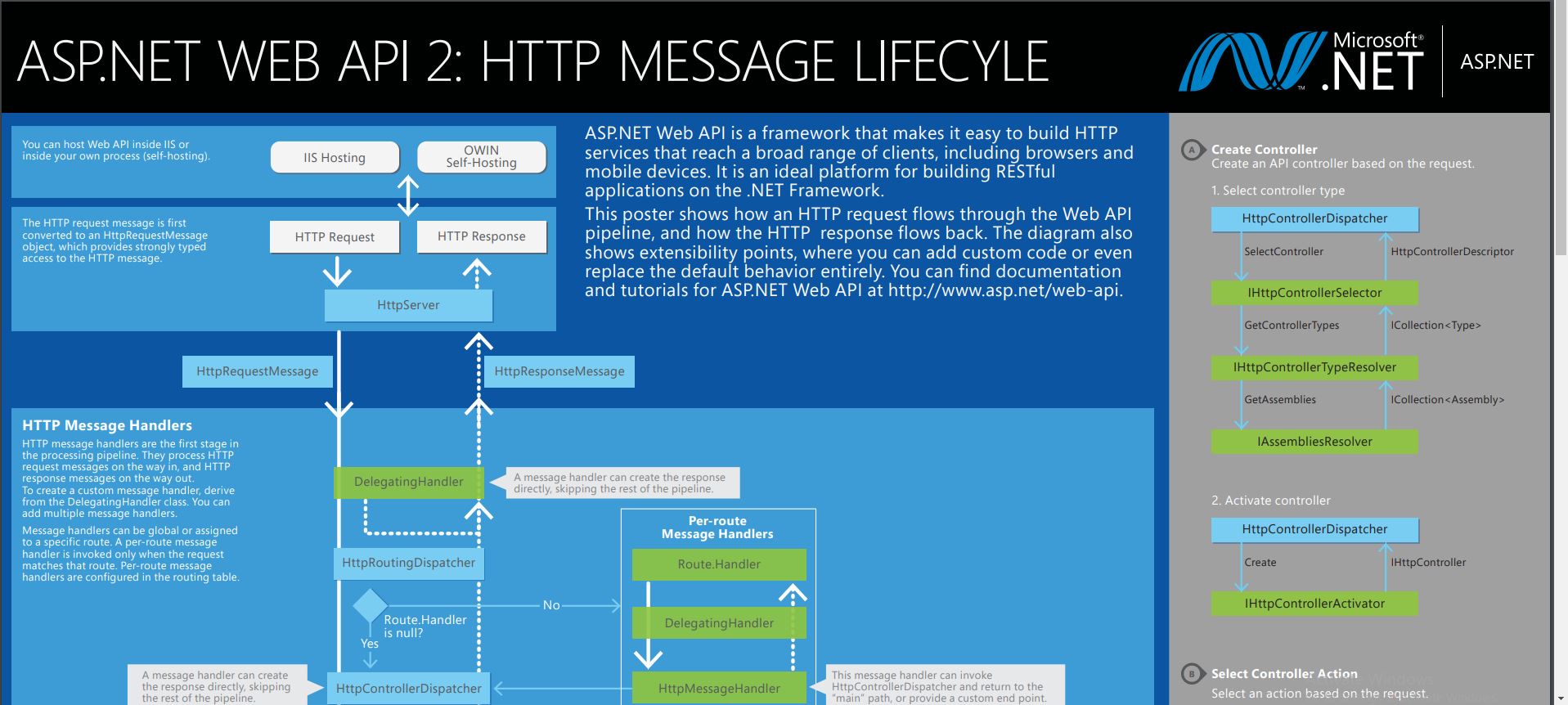
**Message lifecycle (aka “request-processing pipeline”)**

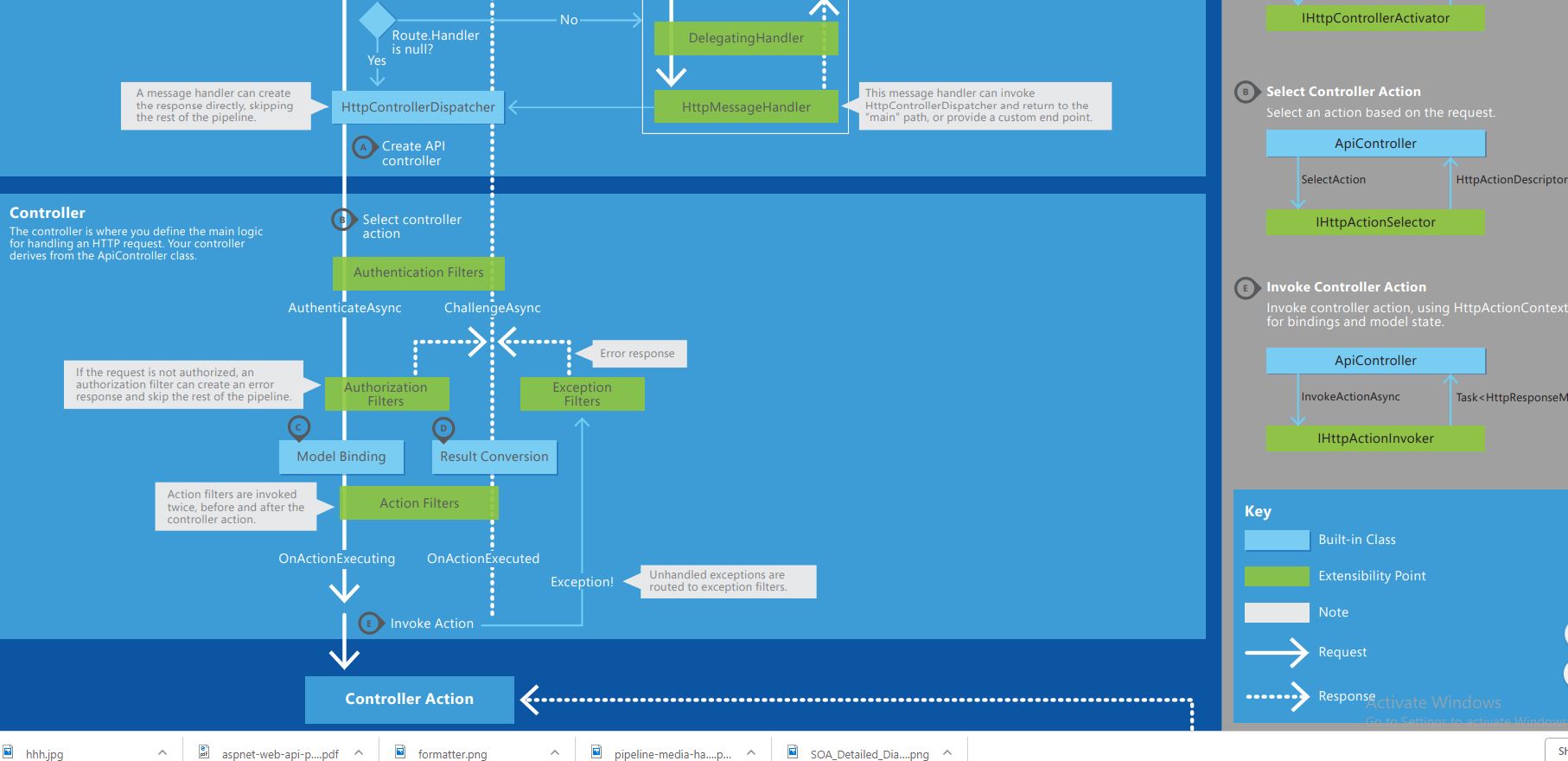
Before continuing, let’s look at a topic that has been briefly and gently introduced a couple of times already.

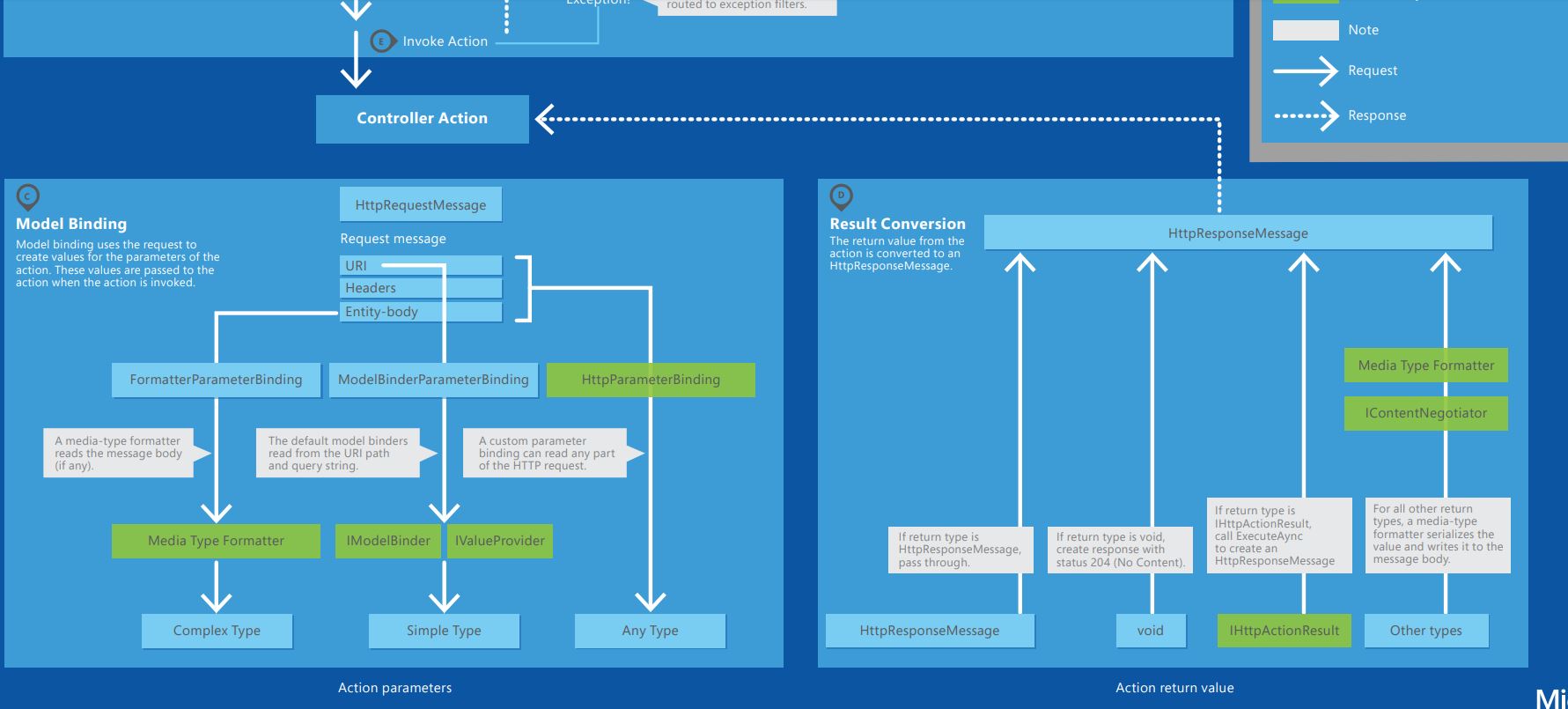
You are probably aware that a request message flows through a *request-processing pipeline* before a response is returned to the requestor.

It’s time to learn more about the *message lifecycle*, so that you can modify or affect a message as it flows through the pipeline. In a few weeks, you will learn how to handle exceptions, and the HTTP OPTIONS method. Both tasks require you to create code that “plugs in” to the pipeline. The pipeline can be modified, customized, and extended.

The image below shows the message lifecycle. Click to open a PDF full-size in its own tab/window.







**Flow from the request to the controller action**

The following is a brief summary of the flow:

1. Web server host handles the request, and creates an HttpRequestMessage object
2. Routing dispatcher module activated for a Web API (or MVC) request
3. Controller dispatcher selects controller, then the action (method)
4. If applicable, authentication happens
5. If applicable, authorization happens
6. Model binding happens, for data in the URI, headers, and entity body \*
7. Controller action (method) is invoked, with the data

\* The deserialize task is done by a *media formatter*.

**Flow from the controller action to the response**

The following is a brief summary of the flow:

1. The result of the controller action (method) is converted to an HttpResponseMessage \*
2. If the result was an error, the built-in error handler is invoked
3. Web server returns the HTTP response to the requestor

\* For a successful result, the serialize task is done by a *media formatter*.

**Summary**

We wanted to leave you with this understanding, because it starts to become very relevant when handling internet media types in a web service.

**Introduction to “media type formatters”**

You have some web service experience working with the JSON and XML data formats. You can reliably accept and deliver JSON content to and from your web service app.

When accepting JSON content into a controller method that handles POST or PUT requests, you have seen that the model binding process *materializes an object from the JSON content*. But how does that happen?

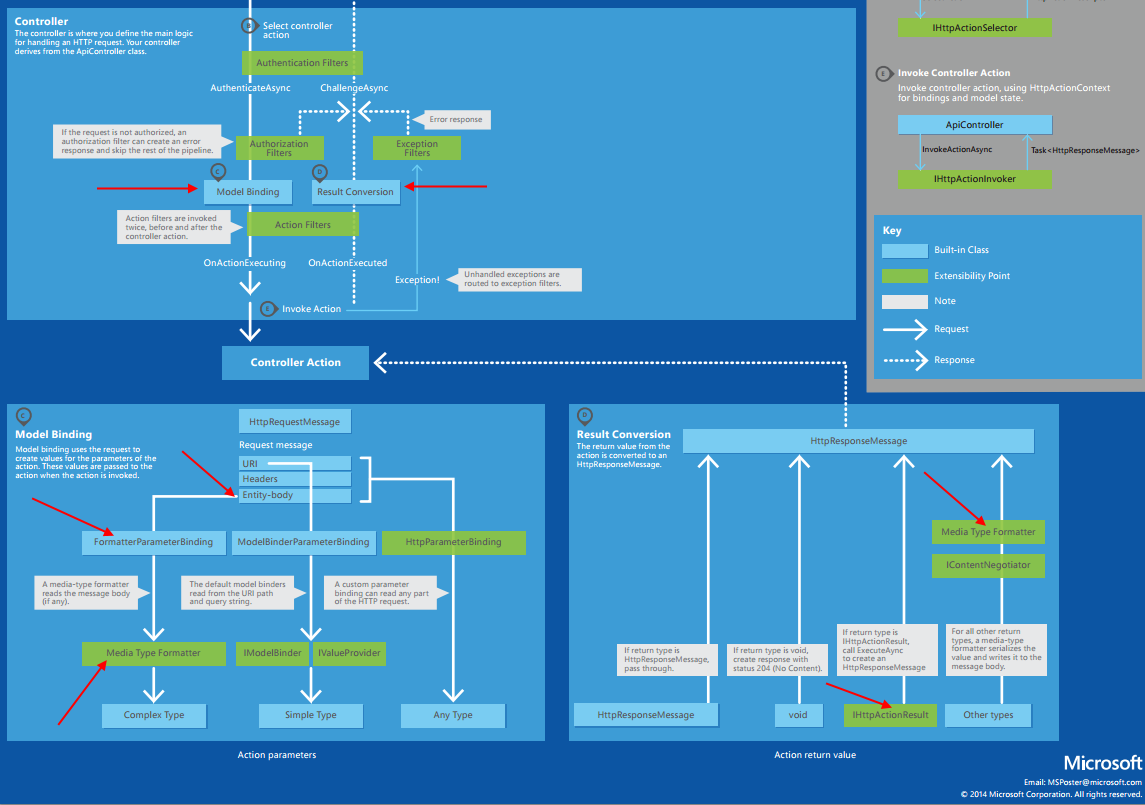
Similarly, when delivering a result (an object or collection) from a controller method that handles a GET request, you have seen that *the object or collection is delivered as JSON in the response*. But how does that happen?

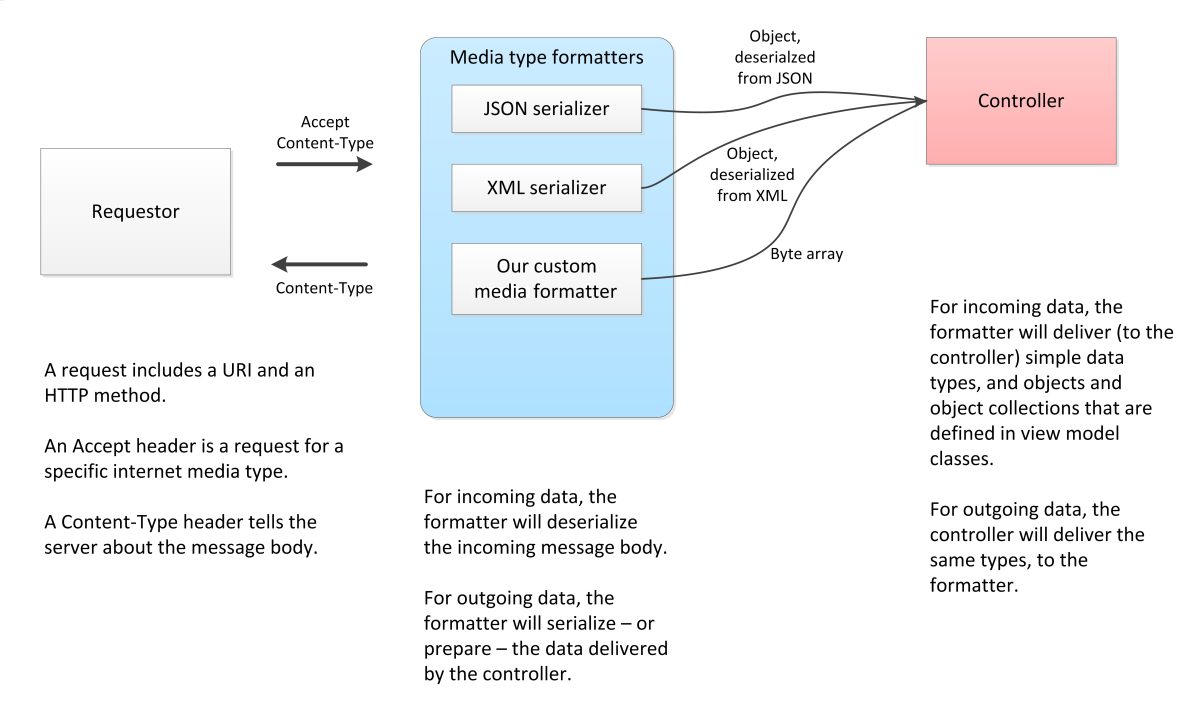
A “media type formatter” is the name of the component that actually does the work. This component is conceptually located between the *requestor* and the *controller action/method*. Every “Web API” project includes media type formatters that handle JSON and XML.

Read [this ASP.NET document](http://www.asp.net/web-api/overview/formats-and-model-binding/json-and-xml-serialization) for more information about JSON and XML serialization.

The image below is an excerpt of the bottom part of the “message handling” image that you studied above.

It has been marked up with red arrows which shows the locations involving the media type formatters in the request-handling pipeline.



 Here is a simplified diagram that shows the request and response flow

**Custom media type formatter**

As you have learned, every “Web API” project includes the built-in media type formatters for JSON and XML.

How do we handle other media types?

We create a custom *media type formatter*.

Incoming: Our formatter will transform (or convert) an incoming media item – which is a byte stream – in the HTTP request’s entity body to an object that we can use in our web service. The HTTP byte stream is transformed into a byte array (i.e. byte[]).

You may recall that your ASP.NET MVC web apps used an [HttpPostedFileBase](https://msdn.microsoft.com/en-us/library/system.web.httppostedfilebase(v=vs.110).aspx" \t "_blank) property in the incoming view model class. This piece of goodness enabled your code to get access to a media item that was sent in a POST request, in an “input type=file” element.

We do not have access to HttpPostedFileBase in a Web API web service. Instead, we create a custom media type formatter.

Outgoing: Our formatter will also transform (or convert) an object in our app – a byte array – to a byte stream in the entity body of an HTTP response.

**Code example: MediaUpload**

Open and study the “MediaUpload” code example, as you continue reading this content. The code example enables us to learn about writing our own media type formatter.

Another code example – *MediaUploadAndDeliver* – will also be studied this week.

We’ll use a simple “book” data entity (books – remember those?), and configure its entity class with properties that hold a photo of the book cover, and the internet media type of the photo (typically “image/png”).

Therefore, this design will work if an entity object has a *single* photo/image property. And conceptually, it can also make sense for that single photo/image to *represent* the object. (In other words, you would *not* use this design if you were building a media-driven photo sharing app/service.)

As a result, it will be possible for a book to have two *representations*:

1. Data representation, as JSON
2. Image/photo representation, as a PNG or JPG image

It will be possible for the requestor to ask for *either* representation.