**DPS907 notes – week 5 – Oct 5**

Hypermedia representations and API design.

**Agenda for the Friday class**

Here’s the agenda for the Friday class:

* Prepare for the hypermedia-driven web API topic
* Building blocks; metadata, links, link relations
* Packaging a response in a hypermedia representation

**Code examples**

In the GitHub [code example repository for this week](https://github.com/peteratseneca/dps907fall2018/tree/master/Week_05), you will see several interesting assets.

The *BetterErrorHandling* code example makes error handling better.

*HttpOptionsHandler* enables requestors to ask about the allowed methods of your web service resources.

*HRIntro* gets you started on hypermedia representations.

**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.

**Error-handling, version 2, minor improvement**

Up to this point in the course, we have been using HTTP status codes and the ASP.NET Web API framework stack trace to communicate error/exception conditions to requestors.

Today, we’ll make a minor improvement to that process. We take advantage of your recent experience in working with a module (e.g. the byte formatter) that plugs into the request-processing pipeline.

In the near future, we will make another minor improvement to the process, by capturing and saving information about the error/exception.

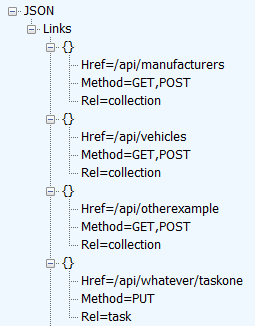
**Studying the changes**

Open the *BetterErrorHandling* code example.

Notice the new class in the ServiceLayer folder. It is initialized in the Register method of the WebApiConfig class.

We have included a throw-away controller, CreateErrorController, that you can use to create (generate/cause) different kinds of errors. Just for fun. Use Fiddler to test your interaction with the controller.

**Configure an *initial URI* for the web service**



In this section, we introduce an idea that provides information about the web service’s resources and functionality to the requestor. The easiest way to understand the concept is to configure an “[initial URI](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)” for a web service project. A “root”, in other words.

The idea is that the web service provider can publish *one single URI* for the web service, and clients/requestors will be able to use that to *discover* all of the available resources, without the need to look at out-of-band or external documentation.

The available resources are rendered as a collection of *link relations*. As a preview of what’s to come, let’s define a *link relation* as a data object, which informs the client/requestor about links to other resources, and actions that can be performed on the current resource.

To publish an “initial URI” for the web service, we must perform these steps:

1. Add a default value “Root” for the “controller” placeholder, in the App\_Start > WebApiConfig class
2. Add a new ApiController class (named “RootController”) that uses the value you provided above
3. Implement a single “get” method, which returns a collection of link relations

Please note that you, as the web service author, will be manually coding and creating these link relations. Soon, you will learn how to programmatically generate link relations.

The *HRIntro* code example was enhanced with this feature.

**Handling an HTTP OPTIONS request**

Modern web services must support the HTTP OPTIONS method.

From [RFC 7231](https://tools.ietf.org/html/rfc7231#section-4.3.7):

“The OPTIONS method requests information about the communication options available for the target resource…  This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.”

This feature enables a requestor to determine what HTTP methods are supported by a URI, *before* sending the actual request. Very useful.

Instead of using GET with a request URI, use OPTIONS with a request URI.

The response will include a response header named “Allow”, with a comma-separated list of HTTP method names that can be used with the URI. It’s as simple as that. (The entity body is empty.)

Does the Web API project automatically handle OPTIONS requests?

Sadly, the answer is *no*. We must write code to do this. For example, in a controller, you could write a method that handles the request. Notice the [HttpOptions] and [Route()] attributes, and a different return type, HttpResponseMessage:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | [HttpOptions]  [Route("api/products")]  public HttpResponseMessage AvailableOptions()  {      // Create a response object      var response = new HttpResponseMessage(HttpStatusCode.OK);      // Initialize the content property      response.Content = new StringContent("");      // Add an "Allow" header, with the allowable methods for this URI      response.Content.Headers.Add("Allow", "GET,POST");        return response;  } |

Do we need to write this kind of code in every controller? Yes.

For every URI pattern? Yes.

That would be annoying.

Is there a better way? *Yes!*

The recommended approach is to create a “message handler”, which is a software component that plugs into the request-processing pipeline. It will intercept and process any request that uses the OPTIONS method. The request is serviced by a single module, and it returns a useful response to the requestor. This is a *much better* alternative to writing code in every controller.

Your professor acknowledges the [post by Jef Claes](http://www.jefclaes.be/2012/09/supporting-options-verb-in-aspnet-web.html) that enabled the handler to get written. Then, [Yao Huang Lin’s post](http://blogs.msdn.com/b/yaohuang1/archive/2012/05/13/asp-net-web-api-introducing-iapiexplorer-apiexplorer.aspx) provided the knowledge that resulted in it covering all kinds of URIs.

**Implementing an HTTP OPTIONS handler**

Instead of writing this handler from scratch, students in Prof. McIntyre’s Web Services course have permission to use the code that’s found in the *HttpOptionsHandler* code example folder. (The folder does not include a complete project – just the source code file that you need.)

Just copy the *HRFormaterICT.cs* source code file into any existing project (into the ServiceLayer folder), change its namespace name to match your project, add it to the Register method body in the WebApiConfig class, and it will just work.

Study the code, to learn how it works.

You’re welcome.

**Hypermedia-driven web API**

Modern web APIs, for public use, must be *hypermedia-driven*.

What does this mean? It means that the response will include:

* the data that you expect, and
* *links* that enable you – the requestor – to determine what to do next.

The web API author should not have to provide out-of-band documentation or knowledge to the user of the web API.

Instead, the user of the web API should be able to “discover” functionality, found in the links, to drive the logic and workflow of their app.

This concept was identified in the *hypermedia constraint* section of Roy Fielding’s PhD (doctoral) thesis, “Architectural Styles and the Design of Network-based Software Architectures”.

Who is [Roy Fielding](https://en.wikipedia.org/wiki/Roy_Fielding)?

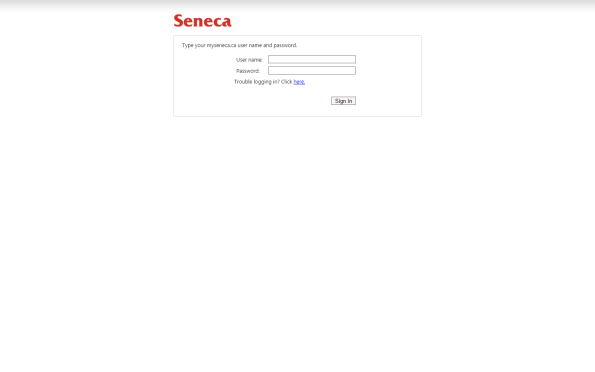
A giant in our industry. We owe him so much.

Oh, and when you look at the list of authors of the RFC 7230 series, he’s the principal author. Then, look back at the now-obsolete predecessors to this series: [2616](https://tools.ietf.org/html/rfc2616), [2068](https://tools.ietf.org/html/rfc2068), and [1945](https://tools.ietf.org/html/rfc1945). He’s on the author list on those, too.

Incidentally, another name on the author list is “H. Frystyk”. That would be [Henrik Frystyk Nielsen](https://en.wikipedia.org/wiki/Henrik_Frystyk_Nielsen). Working at Microsoft, he is one of the developers of and contributors to the framework that powers ASP.NET Web API. Yes, really.

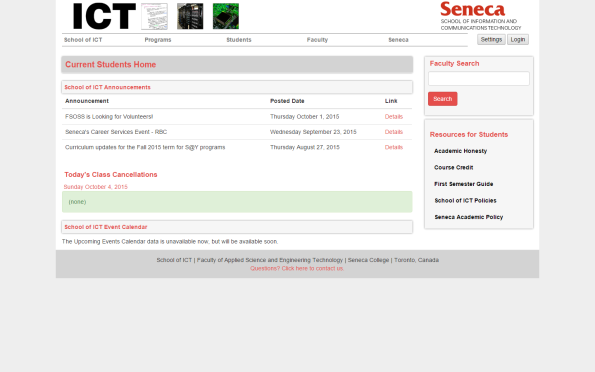
**How can I understand this concept better?**

Study the following screen capture. (Click to open it full-size in a new tab/window.)

[](https://petermcintyre.files.wordpress.com/2015/10/web-page-1.png)

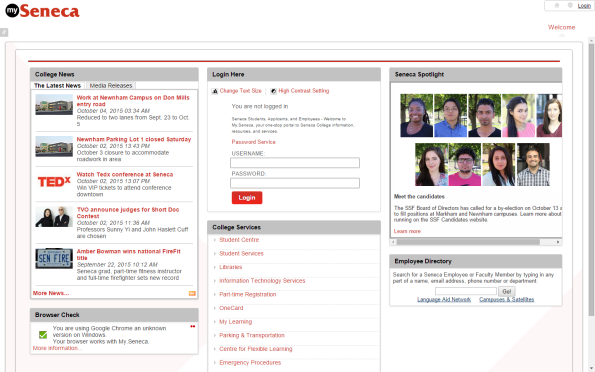
How do you use this page? What is its purpose? What can you do next?

Next, study the following screen capture. (Click to open it full-size in a new tab/window.)

[](https://petermcintyre.files.wordpress.com/2015/10/web-page-2.png)

How do you use this page? What are its purposes? What can you do next?

Finally, study the following screen capture. (Click to open it full-size in a new tab/window.)

[](https://petermcintyre.files.wordpress.com/2015/10/web-page-3.png)

How do you use this page? What are its purposes? What can you do next?

With all three pages – any ANY page you view in a web browser – you innately know what you can do next. You can view the content. You can click links (including the browser’s “back” link). You can enter some data and process that data.

How can you implement this – *what can you do next?* – in a web service?

With link relations. That’s today’s major topic.

**Where can I learn more about hypermedia-driven design?**

Please read the following two documents:

1. Section 5 of Roy Fielding’s [Architectural Styles…](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) PhD (doctoral) thesis.

This is a readable and understandable thesis, but some readers will need more effort to get through the recommended Section 5, “Representational state transfer” (REST). Take your time, and skim when necessary.

Near the beginning, Roy identifies the now well-known REST constraints:

“…multiple architectural constraints are needed to guide the behavior of [software] components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state. These constraints will be discussed in [Section 5.2](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2).”

Yes, the constraints are discussed in Section 5.2. However, the hypermedia-driven concept is better explained in [Section 5.3.3](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_3_3). The last paragraph of the section brings out this idea clearly:

“The model application is therefore an engine that moves from one state to the next by examining and choosing from among the alternative state transitions in the current set of representations. Not surprisingly, this exactly matches the user interface of a hypermedia browser. However, the style does not assume that all applications are browsers.”

2. [REST APIs must be hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven), in which Roy reminds us about some design principles.

**Conflicting opinion – it does not matter**

Before continuing, we should present and discuss a conflicting opinion: *Hypermedia-driven design does not matter.*

Transforming a web service to a hypermedia-driven design takes effort, and must be done correctly. It’s a lot of work.

Consider a scenario where a web service was created as part of a new system’s architecture and design. Further, the web service, and the applications that use it, will be used inside an organization only. In other words, it will not be publicly-accessible. The programming teams that are developing the web service and the applications are unified, or at a minimum, they collaborate during the development process.

In this kind of scenario, the web service and the applications have knowledge of state changes, workflow, object graph shapes, and so on. Documentation is also likely shared, so the teams have a vast amount of information and understanding.

So… Does it matter in this scenario? You could argue “no” in a convincing manner.

Software architect Ben Morris does indeed do this, in his post…

[Pragmatic REST: APIs without hypermedia and HATEOAS](http://www.ben-morris.com/pragmatic-rest-apis-without-hypermedia-and-hateoas/)

Please keep this in mind in the future. You may find yourself in a situation where you face this kind of choice. With a bit more experience in this and related fields, you will be able to bring value to the decision-making progress.

**Data and metadata available to the web API designer and user**

The following list is a brief reminder of the list of *data items* and *metadata* that are available to web service designers and users:

* HTTP method
* URI
* Request headers
* Response headers
* Message body, in general
* Representation (of a resource)

Today, you will learn something about these:

1. Link relations
2. The HTTP OPTIONS method
3. Hypermedia representations, which include additional data that enables a user/requestor to learn about the data that’s needed to add a new resource

**A brief discussion of “links”…**

Web client programmers are familiar with HTML “link” elements:

* a
* img
* link

You also know that the “HT” part of HTML expands to “hypertext”.

So… what kind of data is at the other end of a link in HTML?

Text?

Always?

Or can it be an image, a video, or some other media?

Yes, certainly the data could be *any kind* of internet media type.

Therefore, to help you make progress learning this topic, simply change your understanding (of a link target’s data type) a bit, so that you begin to think of a link as a “hypermedia” link, and not just as a hypertext link.

**How do we use links in a web API?**

In a web API, links are used to tell the requestor about *state transitions*.

In other words, they tell the requestor *what can be done next*.

For example, assume that a requestor asks for a single-item resource (e.g. a product). The web service returns a representation (the data) for that resource.

It would be nice to include data and/or metadata that would tell the requestor whether that resource could be modified, or deleted.

Or, whether it was possible to add a new resource to the resource’s parent/collection.

**Link relations**

The solution is to use *link relations*. Using this solution, the response includes links that can guide the user through state transitions.

This is what Roy Fielding means when he writes that “hypermedia [is] the engine of application state”.

A link is a data object, delivered in-band, *with* the representation. When you define a link in a web service, you must also define its relationship to the current object.

For example, by convention, a single object must include a link to itself:

* The value of the URI (href) will be the item’s URI
* The value of the relationship (rel) will be “self”

If the single object is part of a collection, it must *also* include a link to its parent/collection:

* The value of the URI (href) will be the item’s parent/collection
* The value of the relationship (rel) will be “collection”

This design ensures that the responses from your web service are self-describing.

The [IANA has published a list](http://www.iana.org/assignments/link-relations/link-relations.xhtml) of standardized link relations. You can create your own, but use the standard link relations until you gain more experience.

**Including link relations in a resource model class**

Let’s briefly review the concepts discussed above:

* Modern web services must be hypermedia-driven
* In the response, we include information that helps the requestor understand resource state, and application state changes
* Link relations are used to communicate some of this information

This all sounds a bit abstract. How can we apply the concept? By writing classes that include link relations.

As you go through this section and the next section, open and study the *HRIntro* code example.

**A “link” class**

We need a “link” class that includes string properties for rel, href, and so on.

In the code example, it’s located in the *HRFormatterICT.cs* source code file, in the ServiceLayer folder.

The idea is that we can create an instance of the link class, and then custom-configure it.

**An item class that includes a “link”**

What to do with the link class? Well, we must look at the resource model class for the object or collection in the response. That’s what gets delivered to the requestor, so it makes sense to look here.

In the code example, a new “…WithLinks” class was defined, inheriting from CustomerBase. It simply adds a collection of links to the base class. (It also needs an AutoMapper create-map statement, so remember to do that in the Manager.)

Why a *collection* of links? Well, by convention, an individual item will include a link to self, and a link to the parent (enclosing) collection.

**Packaging items link relations into a customized “hypermedia representation”**

Are we done? No. More information should be added.

We will create a custom “hypermedia representation”. An instance of this class will include the data that we are sending, and some additional information about resource state and app state changes.

Are there standards we can follow? Well, there are no universally-used *de facto* standards. (You will learn more about this issue next week.) For our purposes, we can design our own. Consider the following example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class MediaTypeExample  {    // The default constructor is omitted for clarity      public DateTime timestamp { get; set; }    public string version { get; set; }    public int count { get; set; }    public ICollection<dynamic> data { get; set; }    public List<link> links { get; set; }  } |

Let’s study this class.

**Metadata properties, timestamp, version, and count**

As their name suggest, these properties provide useful information about the representation.

“count” provides an easy top-level and calculation-free way for the requestor to determine how many items are in the collection. A “get one” will return “1” as the value. A “get all” will return zero or more as the value.

“version” is programmer-provided, and can be an easy way to tell the requestor about the version number of the representation.

**The “data” property**

The “data” property holds the response data. Each item includes a collection of links, which gets custom-generated.

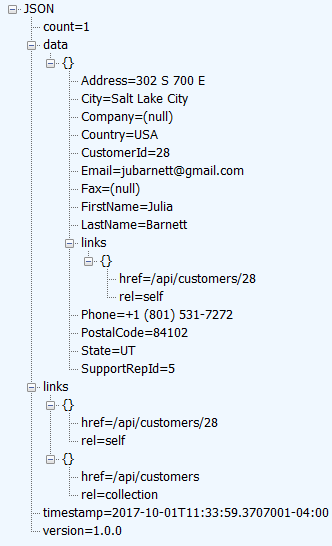
Its data type is [dynamic](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/dynamic), which is a sweet C# feature that provides a useful way to handle different types in one class.

**The “links” property**

The “links” property holds one or more link relations that are relevant for the requested resource. For example, a collection resource will typically have one, pointing to the collection as “self”. Alternatively, an object (one of) resource will typically have at least two. One points to “self”, and the other to the parent (enclosing) collection.

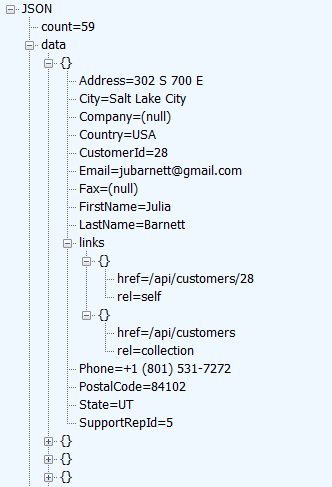
**Example of “get one” customer**

Here’s what “get one” customer could look like:

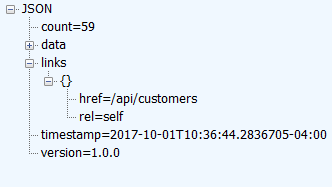


**Example of “get all” customers**

Here’s what “get all” customers could look like. First, the top part…



Next, the bottom part…



**Summary**

Last week, you experienced several new web service concepts and techniques, which you had not seen before when working with ASP.NET MVC web apps. Among them, the most important was content negotiation.

This week, the experience continues. We’re well into a set of topics that cover modern web service development.

You learned a getting-started technique to configure an *initial URI* for your web service, to deliver documentation about your web service in-band, as part of the request/response message cycle.

Next, you began to learn programmatic ways to improve on this technique. Handling the HTTP OPTIONS method is an important topic, as a building block.

Finally, the destination was reached, the one that defines a modern web service (which some people call, whether they truly understand the term or not, a “RESTful web service”). The destination is *hypermedia-driven web API*. It’s a big destination, and you can spend years exploring it (gathering PhDs along the way, as some have done). We explore parts of it in the near future, to equip you for professional-quality work in the web services development field.