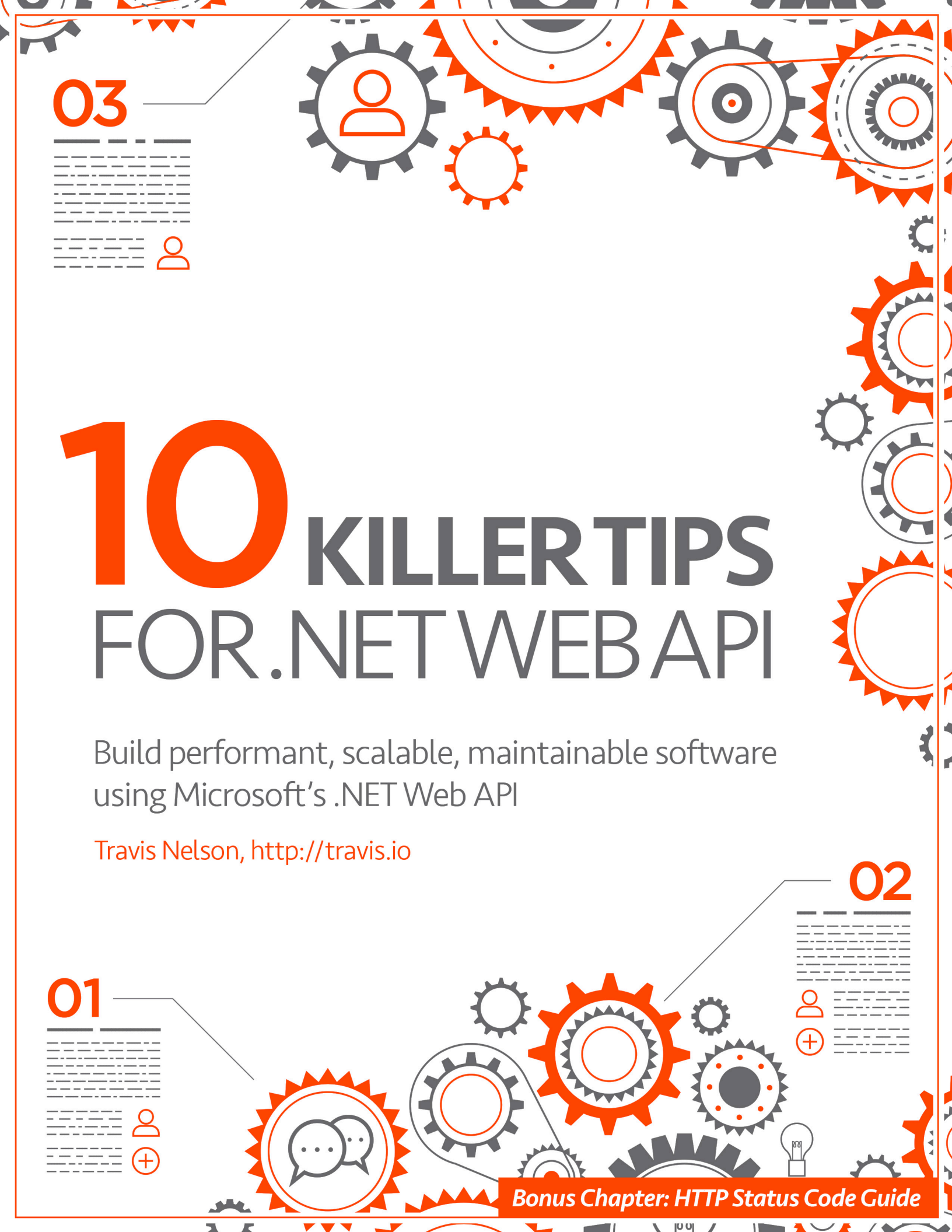

10 KILLER TIPS FOR .NET WEB API

Build performant, scalable, maintainable software using Microsoft's .NET Web API

Travis Nelson, http://travis.io

Bonus Chapter: HTTP Status Code Guide

Table of Contents

About This Guide

This guide is intended to serve as both a series of actionable techniques and as a glimpse into some easy but powerful experiments you can try in .NET Web API to level up your development. The information contained in these pages is a summation of knowledge, experience, and a *lot* of research, and it's my hope that it will inspire you to always be on the lookout for ways to improve your software, just as writing this guide has helped me improve my own. This guide is written in article format and needn't be read in any particular order (to avoid confusion, however, I would suggest reading the words and sentences in left-to-right order.)

Please visit my website at http://travis.io to download the latest copy of this document (this version published June, 2016) and to learn more about me and what I'm up to these days. You'll find more free information just like this, and hopefully some other good stuff too.

Thanks for reading,

Travis Nelson

ONE: Return Errors Like a Champ

Returning Errors Easily and Extensibly

Returning errors in .NET Web API is incredibly easy. The newest implementations of `ApiController` (since Web API 2, at least) provide helper methods for the most common response types, including error conditions. The great thing about these new helper methods is that they're super accessible and available via a simple method call. Here's a quick example:

```
return NotFound();
```

That's it. Whoa. Mind blown.

But this seems unnecessary, right? A helper method for every possible type of return status has a very bad smell to it, so why is it that Microsoft has taken this exact approach here? Well, breathe easy friend, there isn't a helper method for *every* possible HTTP status code. Only the most commonly-used ones are implemented, otherwise there would be dozens. The reason for these new helper methods has less to do with the method itself, and more with what it returns: an `IHttpActionResult` object, which is just one standardized way of returning a response from a Web API method. It also happens to be the current recommended way.

The Old Way of Doing It

Taking a quick step back, there are a few ways to return a response from a Web API method, and the recommended technique has changed as the framework has evolved. In the end each of these techniques equates to some sort of response with headers and content, plus an HTTP status code like 200, 403, etc.; however, some approaches yield more control over the response or your project's architecture.

Old Way #1 - Returning a Model

Simplified examples of Web API suggest returning a strongly typed model which is serialized into the response body, and imply an HTTP status code of *200 OK*. In the case of an error, the best option you have is to explicitly throw an exception, but this feels a bit strange. Here's an example:

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

In this example, an HttpResponseException is thrown with type HttpStatusCode.NotFound, which generates a standard 404 error in the API response. Throwing exceptions is totally normal program flow, right?

Old Way #2 - Returning an HttpResponseMessage

The better alternative to returning just a strongly-typed object (prior to Web API 2) is to take advantage of the `HttpResponseMessage` class, which effectively allows you to separately define the response header information (status code, content type, etc.) and the actual response body. As an example, you can return an error condition like 404, but additionally include a response body. Personally, I prefer to leave error conditions as pristine and non-exploitable as possible by including only the most generic, default data in the response body, but having the option can be useful.

The Right Way With IHttpActionResult

Since Web API 2 was released, the newest recommendation is to return an `IHttpActionResult`, an interface with an abstract method `ExecuteAsync()`. As mentioned earlier, `ApiController` now contains built-in helper methods like `Ok()`, `NotFound()`, `Unauthorized()`, and others which all return some implementation of `IHttpActionResult` and are super easy to use. Here's an updated version of our previous example using the simpler built-in method:

```
public IHttpActionResult GetProduct(int id)
{
    Product p = _repository.Get(id);
    if (p == null)
    {
        return NotFound();
    }
    return Ok(p);
}
```

The benefit here is that it's easy to create your own implementation of the `IHttpActionResult` interface with added functionality; in similar fashion, a few of the existing methods do additional content negotiation and conditional formatting of the response. Additionally, the interface allows for an `async` return type which can improve parallelization in your application, and is covered here in this guide.

In the end, we're still returning a standard response, but an `IHttpActionResult` gives us more flexibility and the option to extend our response and how it plays out. Because we're returning an instance of an interface that we could implement ourselves, we have complete control over the headers and content, and how that information is assembled before sending the response. We likewise have excellent testability, if you're the type who likes to write unit tests.

For a really great example of how the response can be extended (possibly somewhat advanced, but clever) see this GitHub Gist which utilizes an extension method on `IHttpActionResult` and provides the option for cached results and headers: https://gist.github.com/bradwilson/8586562.

Updating Legacy Code Using HttpResponseMessage

It's easy to change a return type to `IHttpActionResult` with little change to existing code you may have which is dependent on `HttpResponseMessage`. Simply change your API method return type to `IHttpActionResult`, and use the built-in converter function `ResponseMessage()` which returns your original message as an `IHttpActionResult` automatically, as below:

```
return ResponseMessage(yourHttpResponseMessage);
```

There, now you can take advantage of using the interface with very little changes to your code.

So, What About Errors Without a Helper Method?

If you want to return an error message with an HTTP status code that does not have an existing helper method, a similar helper method exists on the `Request` object, and the results of using it are effectively the same.

```
return Request.CreateErrorResponse(HttpStatusCode.Forbidden, "Oh no
you don't!");
```

Summary

Web API 2 controllers include great helper methods that make it easy to return error conditions like 404 (not found), 403 (forbidden), and others without the need to worry much about how the responses are prepared. A list of the more commonly used helper

methods is below and should cover you in most situations:

```
Ok()
BadRequest()
NotFound()
InternalServerError()
Redirect()
Unauthorized()
```

Again, each of the above methods simply returns an instance of `IHttpActionResult`, so be sure to use your `return` statement. Creating your own return types is as simple as implementing `IHttpActionResult` and the `ExecuteAsync()` method. Using an action result like this is beautifully polymorphic, and it standardizes responses for Web API regardless of whether the request was successful or not, or whether additional functionality is needed to complete the transaction.

TWO: Gain Control with Data Contracts

Controlling Serialization and Deserialization

The deserialization and serialization of requests and responses is something Web API generally handles out of the box rather elegantly, thanks largely to great libraries like Newtonsoft.Json. That said, sometimes you need a little more control over naming of model properties, or perhaps you have a requirement to hide properties with `null` values. We don't always have the fortune of creating APIs from scratch and full control over method and model signatures, and we don't always have the ability to dictate requirements to our API's client applications.

A Real-World Example

I recently rewrote a Ruby application and API using .NET and Web API, and I'm not sure if you know this, but Ruby developers write crazy things (kidding; I have much respect for anyone out there slinging code.) In typical Ruby convention though, the API model properties used underscores, and you can bet I wasn't about to introduce an underscore into my property names! Because we already had over 100,000 mobile clients out in the world calling our API it was imperative that the API signatures matched verbatim, so the underscore had to stay. Enter Data Contracts, the seemingly forgotten way to gain some

control over your API.

Using Data Contracts

Data Contracts are simple attributes you tag your model and properties with to affect how property names and values are serialized and deserialized (or not.) There are two main components I use, `[DataContract]` and `[DataMember]`, the former providing options for the model, and the latter for its properties.

You Can't Use Data Contracts Halfway

Before we get too far, here's something important to know: If you use Data Contracts for a model, you have to use them for every property on the model that you want included in serialization and deserialization. If you only tag one property with a `[DataMember]` attribute, that is the only property that will be honored, and the other, untagged ones will simply be ignored. You can use this to your advantage in rare cases (but see the nuances below). Conversely if you don't use any Data Contract attributes in your model, all properties will always be included.

The [DataContract] Attribute

`[DataContract]` is an attribute (technically `DataContractAttribute`) which can be applied to a *class*, providing control over its serialization and deserialization. Fundamentally this gives you control over naming via the `Name` property, as in this example:

```
[DataContract(Name = "widget")]
public class Product
{
    // Properties
}
```

Arise, Product, and henceforth you shall be called "widget".

In this example, we're telling the `MediaTypeFormatter` (in the case of Web API requests and responses, but we could manually use a serializer/deserializer) to always serialize and deserialize the `Product` object as `widget` instead. Remember, that goes for both input and output, so make sure your spelling and casing matches exactly.

The [DataMember] Attribute

`[DataMember]` is an attribute (technically `DataMemberAttribute`) which can be applied to class *properties*, providing control over their serialization and deserialization. Much like the `[DataContract]` attribute noted above, this attribute gives control over the `Name` used in serialization and deserialization of object instances, and also includes some other properties like `IsRequired` which instructs the serialization engine that the property must be present when serializing and deserializing, and `EmitDefaultValue` which specifies whether or not to serialize property values if they are default; for example, you may not want to serialize a string if it's empty, or you may not want to serialize an object property if its value is `null`. Here are some examples of each of the above attribute properties:

```
[DataContract(Name = "widget")]
public class Product
{
    // This field is required in serialization/deserialization
    [DataMember(IsRequired = true)]
    public string Name { get; set; }

    // Override the name used for serialization/deserialization
    [DataMember(Name = "creationDate")]
    public DateTime CreatedOnUtc { get; set; }

    // Do not serialize this property if it's null
    [DataMember(EmitDefaultValue = false)]
    public ProductCategory Category { get; set; }
}
```

Fairly straightforward, right? Widgets usually are.

Nuances With Data Contracts

As a "gotcha warning", there are a couple of nuances you should be aware of when using Data Contracts.

- Not all serializers respect Data Contracts; they are simply ignored by some, but sometimes this is configurable.
- Tagging a model with attributes doesn't always guarantee their use; for example, if a tagged class is used as a property type for another class which does *not* use Data Contracts (e.g. a composite or aggregate object), the serializer will not respect the Data Contract attributes.

- Again, Data Contract attributes trigger opt-in mode for serialization, and the introduction of a single Data Contract attribute enacts a requirement to annotate all fields you want included in the output.

Summary

As you can see, Data Contract attributes are an efficient way to control serialization and deserialization of your API models and properties; simply annotate your code for common serialization and deserialization scenarios without the need for code changes or helper methods. Naming of model properties for requests and responses is easy when you are creating new APIs, but requirements to meet specific naming conventions for client requests can often be handled solely through the use of these attributes (for more advanced control over serialization, see the `ContractResolver` chapter in this guide.) Further, Data Contract attributes allow us to adjust the naming of output members without altering our property names or creating property names that don't jive with our style (or worse, cause tools like ReSharper to yell at us.) And as a final benefit, Data Contract attributes can potentially reduce our payload weight and make requests and responses more efficient; converting long in-code property names like `ProductID` or `LastModifiedOnUTC` into shorter names like `id` and `modified` can reduce API bandwidth usage enough to be worth it. API naming conventions don't necessarily need to follow your .NET standards and anything that goes across the wire benefits from some level of optimization, even if only a few percent.

THREE: Don't Name Your Methods Get() and Post()

A Common Scenario

Often when a developer is first introduced to Web API they become captivated by how easy it is to create an API server that responds to HTTP GET or POST requests just by creating methods called `Get()` or `Post()`. These helper methods built into `ApiController` are handy and can eliminate overthinking, but semantically they are vague and non-descriptive. Wouldn't it be better to use method names like `GetProduct()` or `CreateCustomer()`?

I know what you're thinking: "But, we're creating REST interfaces, and action-like names don't coincide with the core concepts of representational state transfer and this is definitely something that Stack Overflow members would yell at me for." Personally, I'm a stickler for convention and best practice, but it's important to consider the distinction between the API's public interface and its underlying code. We want to think RESTfully when designing our consumable API, to be sure, but within our private code we find benefit when writing semantically and complementary to our coding conventions.

Convention-driven practices for naming methods is great, but identically-named methods across many disparate controller classes are difficult to search and navigate through in our

IDE. Want to find the `Get()` method which returns a customer amongst a collection of 20, 30, 50 other ones? It may take only a few extra seconds, but if you did that forty-million times over the lifespan of a project you'd be wasting a lot of time. *Maybe* that's an exaggeration, but at least consider the decision fatigue[1] you'd avoid.

There is a better way. Built into the .NET Framework is a collection of attributes usable in Web API to tag methods to respond to specific types of HTTP request methods. They allow us to use any method names we want in our controllers and simply instruct the runtime which one to use for each incoming request.

The Attributes

The list of attributes here is simple and it's clear what each does. If you're not familiar with HTTP methods and how they should be used in a RESTful API, now would be a good time to start Googling.

- `[HttpGet]` - For GET requests (obviously)
- `[HttpPost]` - For POST requests (obviously)
- `[HttpPut]` - For PUT requests (obviously)
- `[HttpDelete]` - For DELETE requests (obviously)
- `[HttpPatch]` - For PATCH requests (obviously)
- `[HttpOptions]` - For OPTIONS requests (obviously)
- `[HttpHead]` - For HEAD requests (obviously)

Using the Attributes

Applying the above method attributes is straightforward:

```
[HttpGet]
public IHttpActionResult GetProduct(int id)
{
    // Code probably goes here
}

[HttpPost]
Public IHttpActionResult CreateProduct(Product product)
{
    // Code probably goes here too
}
```

Get? Get what? Ooooh, I see, a product. That makes sense now.

In the above example, we named our method according to practices that we're already following, and avoided an ambiguous method signature like `Get(int id)`. This makes our code predictable and self documenting; always a beautiful thing.

Best With Attribute Routing

While these attributes give us control over our method names, they are best complemented by route attributes, which give us additional control over route paths—basically an alternative to route registration. Included in Web API version 2, Attribute Routing places the configuration for route paths right alongside the API method, tying everything together in one place. This is my preference because as the developer I'm able to define not only how my API method is called, but I can also specify a most appropriate API method path. I'll leave Attribute Routing as a research topic for the reader, but a

simple example is included below.

```
[RoutePrefix("api/products")]
public class ProductsController : ApiController
{
    [HttpGet]
    [Route("{id}")]
    public IHttpActionResult GetProduct(int id)
    {
        // Code probably goes here
    }
}
```

Congratulations! You receive your Boy Scout badge for using every form of bracket in a single line!

The above snippet shows simple use of Attribute Routing paired with method attributes applied to a controller and one of its methods. Everything regarding the route is defined here, which some developers find more accessible and intuitive when designing APIs.

Summary

Naming methods with default conventions like `Get()` and `Post()` is helpful in a clinch (like when creating an API that will have a shelf life of a few days), but at best confusing in the long term. These method attributes provide simple but powerful flexibility both for your API and your syntax. Paired with Attribute Routing and the ability to define the most common properties for your API in one place, method attributes make creating Web API services a little more elegant.

FOUR: Crush Performance with Alternate Serializers

Web API Can Be Faster

Web API is already pretty fast. But did you know it's possible to replace the JSON serializer that handles the body content of HTTP requests and responses with different implementations and increase the speed of serialization by 200% or more? In case you're wondering why this small matter is important, consider that serialization and deserialization are the slowest parts of your Web API methods, save of course for network latency.

Speed Matters

We know that speed is important. But how important, really? Google did an extensive study in 2009[2] which concluded that increases in page load times ranging from 100 to 400 extra milliseconds (still under half a second, mind you) resulted in up to 0.6% fewer searches, and that subsequent delays continued to drive that number up to 0.76% over the course of several weeks. That adds up to millions of searches per day, resulting in millions of fewer ads displayed per day, and with Google's primary source of revenue being advertising this was undoubtedly a very expensive study for them to perform! As an aside, shortly after these experiments, Google announced that page load times would be a

contributing factor in search result placement.

Google isn't the only big name doing this kind of research. Similar findings from Amazon, Intuit, and even the 2011 Obama campaign site[3] reveal incredible revenue potential and user retention as a result of fast loading times. Now, of course you're not Google and I'm not Google, but one thing should be strikingly clear: users care very much about load times, even in small increments that we might consider negligible. So, faster is better.

Serializer Alternatives

Out of the box, newer versions of Web API use Json.NET (also known as Newtonsoft.Json) as the default serialization engine. By and large, this is a good thing! The tooling and options in Json.NET are substantial, and the framework is highly performant especially considering that. But it's not the fastest serializer out there and so in high-performance environments it may not always be the best choice, though of course there are myriad other factors to consider. If you want the fastest though, Json.NET isn't it.

New Kid On The Block - Jil

Jil is a newer serialization engine created by Kevin Montrose, employee #4 at Stack Overflow, and is used as part of the architecture of Stack Overflow[4]. That alone should make you want to include it in your project. Jil is highly performant, reliable, and still feature rich. It can serve in a `MediaTypeFormatter` replacement for Json.NET to serialize and deserialize Web API method requests and responses, and a library with instructions for doing that can be found here: https://github.com/bmbsqd/jil-

mediaformatter.

To replace Json.NET with Jil as the default `MediaTypeFormatter`, you can add the following line to your *Global.asax* `Application_Start()` method, or anywhere else you have bootstrapping code:

```
config.Formatters.Insert(0, new JilMediaTypeFormatter());
```

It's also possible to leave Json.NET as the default formatter and use Jil selectively, for example per controller, by creating a custom controller attribute like this:

```
public class HighPerformanceSerializationAttribute : Attribute,
IControllerConfiguration
{
    public void Initialize(HttpControllerSettings controllerSettings,
                           HttpControllerDescriptor
controllerDescriptor)
    {
        // Register the Jil MediaTypeFormatter for this Controller
        controllerSettings.Formatters.Insert(0, new
JilMediaTypeFormatter());
    }
}
```

You could then tag your controller class with the attribute like so:

```
[HighPerformanceSerialization]
public class KittyCatController : ApiController
{
    // Controller methods probably go here
}
```

The biggest downside to using Jil (and the reason it makes most sense to include it as an

option, without replacing Json.NET) is its limited configurability. Jil's configuration options should suffice for a majority of needs, but are limited to things like camel-casing, ignoring null values, and date formatting. Outside of that, Jil lacks much of the rich options of a more robust framework like Json.NET, which uses abstractions for a majority of its internal components and provides an API allowing you to replace and customize it in incredible ways. But it's *fast*.

Optimization Level Up - Protocol Buffers (AKA - Protobuf)

For all of its internal services, Google uses a custom serialization framework called Protobuf that is incredibly fast and takes advantage of strong compression algorithms. Because of the optimizations built into it, Protobuf is widely considered the fastest serializer available, and lucky for us, Google has made it available for download and use in our projects. The only downside is that Protobuf offers only binary serialization (i.e. not JSON) making it just slightly more complicated to implement. There are, however, libraries available for many major server-side languages and at the time of this writing, an official version is in development for JavaScript.

I wouldn't suggest Protobuf as the default `MediaTypeFormatter` for your public API, but instead as an alternative one that can be requested during content negotiation (`Accept` and `Content-Type` headers: `application/x-protobuf`). Protobuf is also an excellent choice for manual serialization and deserialization, for example when storing data into table or blob storage, and for communication between internal services.

Summary

I debated introducing other serializers here, but I don't really see a need. For JSON serialization Jil is a strong framework that is growing in popularity, and for binary serialization there's nothing that beats Protobuf. If you'd really like to look at other options, the ServiceStack.Text framework is very popular and highly performant but has recently become a paid option with limited free usage (https://servicestack.net/text), and fastJSON is a strong contender and a bit of an underdog with some raving fans (http://www.codeproject.com/Articles/159450/fastJSON).

With the ability to swap the default Web API serialization framework with a simple line (or two) of code, it makes little sense to avoid it. If you're worried about replacing the default serializer and impacting your existing API, the migration path is straightforward; simply include the new option in the `MediaTypeFormatter` collection as a non-default option (if it can be triggered via an `Accept` header), or add it to the formatter collection of specific controllers as shown above so you can refactor one controller at a time in a phased approach.

Depending on your setup and on traffic against your API, you may see a significant performance gain by using a non-default serializer, particularly if you have the ability to utilize Protobuf and serialize to binary.

FIVE: Hack the Json.NET ContractResolver for Maximum Control

Convention Is Nice, Until It's Not

Web API gives us a lot out of the box. In a brand new, basic Web API project we have access to excellent default formatters, automatic content negotiation, and the use of MVC's relatively logical conventions for routing, request and response methods, among other things. It's a pretty sweet deal, and right in line with what Microsoft has been doing well for a very long time—making it quick and easy to churn out functional line-of-business applications that fit mostly within a particular mold. Of course we have great extensibility, but the ease of creating a working RESTful API in .NET is pretty spectacular.

Sometimes, though, the mold into which our API has to fit differs slightly from what Web API offers as a default. Whether it's matching a legacy feature or API signature for backwards compatibility, or your manager happened across the Stripe.com API documentation and insists yours must follow its conventions, or someone in your sales team caught wind of your shiny new API project and has strong recommendations about features it absolutely must have because all clients simply require them, public API development is rarely so straightforward. The more stakeholders your API involves, the

more likely it is that someone is going to gently urge (i.e. "require") you to implement a feature that involves more than a few lines of standard Web API code.

One area of strong contention, and for good reason, is that of request and response signatures. By this, I am referring specifically to the payloads and properties of our API methods and how they may or may not change based on the content sent to them. There are myriad ways to control what data we send and receive in our Web API methods, but one which gives us incredible control is that of the `ContractResolver`, the feature responsible for serialization and deserialization in Json.NET. Given the default use of Json.NET and its awesome extensibility we have great control over what goes in for serialization, and what comes back out as our endpoint response.

The Json.NET ContractResolver

Json.NET provides several extensibility points via `JsonSerializerSettings`, a class usable in construction of a `JsonSerializer` object, or when using `JsonConvert` wrapper methods for serialization and deserialization. Examples of these extensibility points include things like date formatting options, `null`-value handling, reference loop handling (what to do during serialization when an object references itself), and more. One property of the serializer settings that opens up a world of customization is that of the `ContractResolver` (implementation of `IContractResolver`) which gives finite control over the entire serialization and deserialization process. Here you can write custom code to make fine-tuned adjustments to the output. Want to add a property to your API response with the current UTC date and time? Or order properties alphabetically?

How about only serializing properties if they start with the letter "Q"? All of this is possible through the use of a custom `ContractResolver` provided to Json.NET.

Example of a Custom ContractResolver

Following is an example of a custom `ContractResolver` (it's usually easiest to inherit from the base abstract type and override/utilize its virtual methods, rather than re-implement the entire `IContractResolver` interface):

```
public class CoolCatsCustomContractResolver : DefaultContractResolver
{
    protected override IList<JsonProperty> CreateProperties(Type
type, MemberSerialization memberSerialization)
    {
        var properties = base.CreateProperties(type,
memberSerialization);
        foreach (var p in properties)
        {
            p.PropertyName = "meow_" + p.PropertyName;
        }
        return properties;
    }
}
```

Just go ahead and copy-paste this into your project. It's production ready.

In the example above, we've created a new implementation of an `IContractResolver` usable in our JSON serializer. The `CreateProperties()` method is responsible for creating the property names that are output in the resulting JSON string (each is actually an instance of `JsonProperty`, itself also having many extensibility points). Here we're

just prepending the string "meow_" to each property name resulting in names like `meow_name`, `meow_totalPrice`, etc.

Going One Step Further With ValueProvider

For further control of the output, you can create one or more custom value providers for the properties created by the `ContractResolver`. Each instance of `JsonProperty` has a field called `ValueProvider`, which it uses for value retrieval/conversion, but the logic here can be customized to return most any value. The interface for the value provider has a method for getting a property's value which can then be serialized, called `GetValue()`, as well as a method for deserializing into a property from the JSON string value, called `SetValue()`. In either case you have full control over the final result.

Using a Custom ContractResolver

There are a few ways you can use an `IContractResolver` when working hands-on with Json.NET. Below is an example for doing so using the `JsonConvert` static wrapper methods. Note that in order to do this, you'll need to pass in the contract resolver by way of a `JsonSerializerSettings` object since no overload exists which accepts it on its own.

```
var resolver = new CoolCatsCustomContractResolver();
var settings = new JsonSerializerSettings {ContractResolver =
resolver};
var result = JsonConvert.SerializeObject(widget, settings);
```

What About The Web API MediaTypeFormatter?

Fortunately, using these contract resolvers across the board for our API methods is straightforward, and an interface exists for adding them to our `MediaTypeFormatter`, provided it's an instance of `JsonMediaTypeFormatter`. In the code below, which would go in your application startup code, we're simply querying our application's collection of formatters for the existing Json.NET formatter, and explicitly setting the `ContractResolver` property for the underlying serializer instance (by way of the `SerializationSettings` property, which is the only public interface.) This instructs our serializer to always use this contract resolver for all responses.

```
// Here, config is an instance of HttpConfiguration
var formatter = config.Formatters.FirstOrDefault(f => f.GetType() ==
typeof (JsonMediaTypeFormatter)) as JsonMediaTypeFormatter;
if (formatter != null)
{
    formatter.SerializerSettings.ContractResolver = new
CoolCatsCustomContractResolver();
}
```

Other use cases for contract resolvers in Json.NET follow similar patterns, such as when manually using a `JsonSerializer`, or when overriding the formatter at the controller level, which can be done using an `IControllerConfiguration` implementation and a custom attribute (example: http://stackoverflow.com/a/22464964/638087).

Summary

It's no wonder Microsoft deemed Json.NET a fantastic default for serialization and deserialization in Web API. Out of the box, Json.NET provides reasonable defaults, excellent speed, and easy configuration. When more flexibility is needed in serialization, the ability to override the tool's default `ContractResolver` paves the way to manipulate results in almost any way you choose. You can override property names, or choose whether they'll be included in the result based on custom logic, and more. With the addition of custom `ValueProvider` instances, you're also given direct control over the values of serialized properties. All of this happens in typical, standard JSON fashion too, meaning you're guaranteed valid JSON in the end after the actual serialization takes place. The biggest benefit here is that this manipulation of the results happens outside of your model classes, which lets you keep them pristine and avoid introducing (or removing) properties when it doesn't make sense to do so.

SIX: Don't Test With a Browser!

API Development Tooling

> *This tip is included here based on my experience with how common it is for developers to test APIs during development using a web browser. If you'd like more in-depth and technical information about this topic, I wrote about it here: http://travis.io/blog/2015/10/13/how-to-get-aspnet-web-api-to-return-json-instead-of-xml-in-browser/*

Web API was created to serve as a fundamentally simple way of creating public APIs in .NET that was very accessible to developers. For most first-time API developers though, the question of exactly how to test an API seems a bit daunting. Because we're used to creating software that runs in a browser, the go-to tool for testing code is typically a browser as well.

Web developers using .NET are used to HTTP requests and serving up content, and often using headers in requests and responses, but for those who have never worked directly with an API, there seems to be a tendency to do a lot of testing within a browser like Google Chrome. While there's often nothing wrong with this, you simply don't have the control of a more dedicated tool built specifically for testing APIs. The tools range from free applications and browser plugins to "holy crap this pricing doesn't even make sense" enterprise offerings, but the key takeaway here is the same: don't use a browser to test

your API!

Browsers Suck for API Testing

First, a story: The first time I created a RESTful API many years ago, I created HTML forms to POST to it. It sucked. Now I use real tools. It doesn't suck anymore. *The end.*

Browsers are great for simple requests to an API. That is, if you want your API to return content in XML format, or if you want to hand-craft HTML forms just to test POST requests. Browsers like Google Chrome send an `Accept` header that tell the server what format to use when sending the response, and the default for browsers is generally something like `text/html`, `application/xhtml+xml`, `application/xml`, or something along those lines. Both a blessing and a curse, Web API can return XML, and as a default response to a request from a browser, does so.

What If You Like JSON Better?

So what if you want a JSON response instead? Use a browser plugin to modify the `Accept` header? Sure you could do that, but it's a hack. Unfortunately the most-often recommended solution online is to just remove the XML formatter from Web API, forcing the return of JSON data. This is absolutely the wrong way to do this! One of the most beautiful features of Web API is its ability to inspect the `Accept` header and return a response that matches one of the requested formats along with a `Content-Type` header. This is part of the content negotiation process and it exists for a reason. What this means is that our API can return XML *or* JSON, and it's up to the client to decide. So why remove

the ability to return XML entirely and cripple our API? Because we're using the wrong tool to test our API, that's why.

What You Should Use to Test APIs

There are countless tools on the market for testing APIs, but there are a few that I prefer over the others. Depending on the depth into which you want to test, some tools provide lower-level functionality but by and large, there's really one tool I have running full time when working on an API, and that's Postman.

Postman

Over the years, tools for testing APIs have come and gone. I prefer a tool that's simple to use and can cover all of my use cases, and I tried many until I eventually found Postman. There's no reason to continue your search. Postman is what you are looking for (full disclosure, I have no affiliation with Postman though I have exchanged emails with the founder in the past; I honestly feel it's the best product on the market.) Postman has a great, simple interface and tools for saving (and syncing) profiles with API requests and data, so they're accessible on any computer you're signed into. There's also cloud-synced team plans at a cost, but it's reasonable if you make Postman a part of your team's standard toolkit.

Using Postman is straightforward. Simply click the tab to create a new request, select the HTTP method (GET, POST, etc.) and enter a URL, then add in any header and body data you need. You can save the request, make copies of requests, and even organize saved

requests into folders and sub folders, making it easy to keep track of the requests you've run and duplicate them in the future. There's also tooling built into Postman that allows you to automate requests in particular order, including some output of statistics.

Postman is free, and is available for Windows and Mac at https://www.getpostman.com/.

Summary

Browsers are meant for browsing the web, and API testing tools are used for API testing. While it's certainly possible to use a browser to perform simple GET requests to verify the existence of an API endpoint or its data at a high level, just remember that you don't quite have the control over headers, request/response types, and content negotiation that you do with a tool designed specifically for that purpose. Postman is my tool of choice for this purpose but there are many others, and I encourage you to check them out. But please, for the love of kittens, don't rely solely on a browser to test your APIs.

SEVEN: Only Deserialize When Necessary

Is Request Deserialization Always Necessary?

Consider this for a moment: an HTTP POST comes into your API containing a payload of information to create a resource in your application. Let's assume it's a new "add to favorites" action for a product in your store made by a customer, and that request is verifiably made from your customer's account. Provided that we can be certain our request is valid and authorized, the common approach here would be to deserialize the request body into an object, possibly map it to a domain or data entity type, adjust its properties as needed, and then save it to our data store, finally returning a serialized response stating that the operation was successful. In a majority of cases this is completely acceptable, relatively fast, and supports our use case of creating the "favorite item" resource perfectly. It's a simple enough request and one that's not unlike what you probably do every day, but let's take a moment to analyze exactly what kind of request this is.

In this situation, we're forcing the application to process our request and create the resource immediately, and in so doing we're hijacking an HTTP request and server resources to do it. But how critical is this, exactly? Is it absolutely essential that we add the

new resource "right now" or is it sufficient for it to be added in a few seconds or even a minute? What about in high-traffic websites with millions of product requests per day where API traffic is already high and throughput for actual sales should take priority over traffic for "add to favorites" extras?

Questions like this might seem specific to optimization scenarios that companies like Amazon or others might face, but the crux of the matter is the same for many of your API scenarios; is real-time processing necessary, or is near-real-time processing sufficient? More specifically, is it essential that we deserialize into an object and process the payload now, or could we just store the request as a native string and throw it into a queue, and have a background process handle it at some point in the near future?

Deserialization is (Relatively) Demanding

The process of serializing and deserializing resources is relatively complex and computationally expensive when compared to API requests and responses as a whole. The default behavior of a Web API method assumes serialization and deserialization for all requests and responses, but this isn't a requirement. It's possible to get a hold of the raw request body as a string and process it or store however we see fit, thus avoiding deserialization altogether and ergo removing a large portion of the processing burden.

In our example of a user adding an item to his or her favorites list, we could accept the request body as a string and immediately store it to a queue, table storage, or database in native format. At this point, we've done nothing with it, however we can choose to deserialize and process it in the future via a background worker role separate from our

API. This could happen mere seconds from the time of the request, or even minutes or days if the use case permits it. We've now delegated non-essential processing to a secondary process, and freed up a substantial portion of our processing to handle more important API requests. Streamlining API traffic is a good thing, and fast response times keep customers happy.

Remember, the goal here isn't to avoid deserialization entirely; the goal is to deserialize the request data into an object at a more convenient time, and to delegate that task away from our API threads.

How to Avoid Request Deserialization

Grabbing the request body as a simple string (its native format) is simple, provided you have your API routes configured properly. Here's an example of how this could be done:

```
[HttpPost]
public async Task<IHttpActionResult> AddWidget()
{
    var result = await Request.Content.ReadAsStringAsync();
    // Do something here with the result
    return Ok();
}
```

Note in the code above, no argument is specified in the method for the body parameter. Including an argument here would force deserialization into an object which is exactly what we *don't* want. In this case, we are hinting to Web API that we'll handle the request body ourselves and we do so via the `ReadAsStringAsync()` method on the request

content, giving us the request body as a simple string. Super efficient!

We can now save that string to a queue and later handle it in a worker process. Doing so requires manual deserialization and should include some validation (as you'd do on your API endpoint anyway), but this gives us control over when to process this data and removes the burden from our ever-important API processes.

Summary

The default, and most-often taught way to use Web API consists of deserialization during HTTP requests as a matter of course. As this is an expensive process, it's important to consider whether we need immediate, real-time processing in our applications, or if near-real-time processing is sufficient.

Opting to use the raw content in a Web API request in our application gives us flexibility and the ability to delegate processing of requests to background workers or even other services. Web API endpoints scale best when more burdensome tasks are removed from the pipeline and when threads are conserved for lighter processing. Plus, lightening the load on your API servers makes it easier to scale horizontally (adding more small servers to your application), as opposed to vertically (increasing server size) and provides control over costs when using auto-scale techniques. Granted you'll still deserialize that string at some point, but the option to choose when will allow you to create a more flexible architecture with better control over your servers and your costs.

EIGHT: PATCH Like A Pro

Support For HTTP PATCH Sucks; Do It Anyway

Unless I'm alone in this, the use cases for the PATCH method in HTTP are plenty. Partial updates are a common need in our APIs, and yet support for them in Web API has been limited, forcing developers to implement their own techniques. This generally involves custom endpoints for contextual patching using strongly-typed models (for example, a PATCH endpoint for updating a users's email address, and another for their preferences), or a single endpoint accepting a dynamic type like `JObject`, paired with some custom logic for mapping the values to an existing object. Another alternative exists in Microsoft's `Delta<TEntityType>` class, a near-perfect fit having its own set of problems, not least of which being that it's baked into *System.Web.Http.OData* making it a less-than-ideal candidate for our needs.

None of the approaches above are *bad*, per se, but likewise none use PATCH as it was intended. Unfortunately the options for sending standards-compliant partial updates are limited at best, and I've yet to see an option that handles it gracefully across multiple Content-Type options. There are options for partial updates, however, but every one [that I've seen] requires some sort of compromise. Because I typically work with JSON, I opt to

support PATCH operations only for JSON, but technically you could implement a similar technique for XML or any other format. The silver lining here is that the method I'm outlining below supports a proposed and known standard for HTTP PATCH using JSON data, called JSON Patch. The implementation uses third-party libraries for .NET versions prior to 5.0, but in versions 5.0 and up these standards will be built into the .NET framework, so code here is majorly future proof.

Introducing JSON Patch

JSON Patch is a proposed IETF standard which outlines a data structure for sending PATCH requests over HTTP. In most developer implementations of PATCH, these requests are simple key-value pairs that describe new values for model properties. This approach is simple and effective, and I don't find anything wrong with it. HTTP purists, however, will be quick to tell you that PATCH requests should define not only *what* properties should change, but also how they should change and in what order, whether they should be added or removed on the resource, and also whether the resource or its properties should be moved as part of the request. For 99.9% of our use cases this is overkill, but fortunately we can find a happy medium by implementing our PATCH request properly while keeping it simple.

At its core, a JSON Patch request might look something like this:

```
PATCH /api/widgets/10 HTTP/1.1
Host: example.org
Content-Type: application/json-patch+json

[
    { "op": "replace", "path": "/name", "value": "New Widget Name" },
    { "op": "replace", "path": "/price", "value": 79.95 }
]
```

In the above request body, we're passing a collection of operations to be performed in sequential order ("sequential" meaning within the current operation, but the entire request should still be performed atomically in a single pass-or-fail transaction.) Keep in mind this format is specific to JSON Patch, but it meets all of the requirements of the HTTP specification for PATCH requests. As you can see, we're replacing the value for the `name` and `price` properties and new values are provided in the request payload. And you don't necessarily need to fully understand the syntax here, since JSONPatch (the .NET implementation I'll be covering here—note the name) handles this for you, as you'll see next.

Using JSONPatch for .NET

> *Going forward, I'll be referring to the language-agnostic, original specification as "JSON Patch" (with a space) and the .NET implementation and NuGet package as "JSONPatch" (without a space). This might seem confusing but I just wanted to make it clear this is intentional and not a typo.*

As with any good third-party library, JSONPatch can be installed via NuGet here (https://

www.nuget.org/packages/JsonPatch/) via `Install-Package JsonPatch` in Visual Studio's Package Manager Console. Doing so will add all of the functionality you need right into Web API. For your client applications, a number of projects exist for languages like JavaScript, Ruby, Python, PHP and more (http://jsonpatch.com/) and these can help you complete your integration of JSON Patch when your requesting application isn't C#. From here, using JSONPatch is simple. The GitHub repository README file (https://github.com/myquay/JsonPatch) has excellent instructions so please refer to them as the source of truth, but the steps are summarized below.

Include the MediaTypeFormatter

First, include the `JsonPatchFormatter` in your application so Web API can use it to process PATCH requests targeted at it. This is done in your *Global.asax* file or other bootstrapping code called from `Application_Start()`.

```
public static void ConfigureApis(HttpConfiguration config)
{
    config.Formatters.Add(new JsonPatchFormatter(new
JsonPatchSettings { PathResolver = new FlexiblePathResolver() }));
}
```

Inception. Instantiation. Instantption.

Note in this example, I'm passing a new instance of `JsonPatchSettings` into the constructor and manually specifying the resolver. This `FlexiblePathResolver` supports the use of Data Contract attributes which provide control over property names in serialization and deserialization. If you're not familiar with Data Contracts, you can read

more about them in the Data Contracts chapter of this guide.

Deserialize and Update Using JsonPatchDocument

Included in the JSONPatch library is an interface called `JsonPatchDocument<TType>`, which handles patching logic sent in the PATCH request and applies those changes to a model that you provide it. To use this interface, just include it in your method as the body parameter, and it will handle everything for you. Here's an example:

```
[HttpPatch]
public IHttpActionResult UpdateWidget(int id,
JsonPatchDocument<NWidget> widgetPatch)
{
    var widget = repository.Get(id);
    widgetPatch.ApplyUpdatesTo(widget);
    return Ok(widget);
}
```

Well. That's probably the easiest thing I've ever done.

As you can see, using `JsonPatchDocument` is as simple as including it in your method (as the body argument, optionally providing the `[FromBody]` attribute) and calling its `ApplyUpdatesTo()` method, passing an instance of the type you're patching. JSONPatch handles the rest for you, and processes the steps you provided in the request to update the model.

Configuration and Nuances for JSONPatch in .NET

There's really not much more to using JSONPatch outside of some basic configuration. In the code above I instantiated the formatter using a non-default resolver which supports

property naming and serialization options via Data Contracts. JSONPatch provides a few different formatters which you can read in the GitHub repository's README file.

It's also worth mentioning that while the JSON Patch specification outlines operations like "add" and "remove", those operations are handled in JSONPatch in a way consistent with the strongly-typed languages of .NET. For example, "add" will only support setting a property value if it exists but has a value of `null`, and "remove" will only set a property value to `null`. Remember, JSON Patch is not unique to .NET and the specification is used in other languages as well, some of which support adding and removing model properties dynamically. Again, more information about this can be found in the documentation for JSONPatch.

Summary

The JSON Patch specification was created as a standard but flexible way of handling PATCH requests via HTTP. Because the actual specification for HTTP Patch states that requests should contain instructions describing how a resource should be modified, and that such requests should be handled in an idempotent manner, the specification helps ensure consistency across languages both in API design and actual implementation. The JSONPatch library for .NET outlined here provides excellent support for the specification and does so in a future-proof manner; Microsoft has already added support for it in .NET 5, even largely using the same class and member names, so migration to officially-supported versions should be straightforward with minor code changes.

NINE: Use Asynchronous Methods in Your API

Easy Performance Gains

Remember the first time you installed an SSD into your computer, or purchased a computer with one installed. The introduction of SSD storage devices marked one of the most significant, cost-effective ways to improve performance on any computer. Considering SSD devices provided the exact same interface as their predecessors but provided performance boosts of 2x or more, moving to them as quickly as possible (and as budget allowed) was a no brainer.

Such is the same with `async` methods in .NET. For very little overhead and some moderate (usually) refactoring, these methods provide incredible gains in processing when dealing with I/O-bound processes, or those delegated to services over the wire where network latency is unavoidable. Microsoft introduced `async` syntax in .NET 4.5 to make asynchronous accessible to developers in a way that abstracts away the complicated management of parallel processing, but still provides all of the benefits. One of the big wins here is the ability to use `async` methods in Web API, which allows us to write methods which meet client expectations for requests and responses, but more efficiently use server resources; this [usually] simple upgrade can provide tremendous wins in terms

of performance and scalability in our API.

Where Async Wins

The details of how this all works are well documented online, and of greater importance is understanding where you'll find the biggest wins when using asynchronous operations. Again, this is primarily when using I/O or network-bound logic since these are typically the slowest operations we deal with, and Microsoft has already built excellent support for these areas right into the .NET Framework. Some examples of these supported APIs are listed below:

- Web/Network Access: `HttpClient`, `SyndicationClient`
- File and I/O: `StorageFile`, `StreamWriter`, `StreamReader`, `XmlReader`
- Images: `MediaCapture`, `BitmapEncoder`, `BitmapDecoder`
- Entity Framework and LINQ: Version 6 and above provides new methods like `SaveChangesAsync()`, `ToListAsync()`, among others
- Azure: Many of the Azure SDK APIs provide asynchronous versions of methods for storing to blob, table, queue, and more.

What Async Doesn't Do

I'd be remiss if I didn't touch on this, since I'm suggesting using asynchronous API endpoints. That is, I'm recommending use of the `async` keyword on Web API methods for GET, POST, etc. Asynchronous methods provide some potentially huge benefits, but they aren't the following:

- **Async does not change how client applications work with your API.** There's no

fancy magic that makes the actual HTTP endpoint respond differently to clients, or work better with any particular client library, or stream or `yield` results to the calling application. To clients, the results will effectively be the same, though potentially faster (I guess if you have a client application cluttered with race conditions, this could be a problem.)

- **Async doesn't automatically execute the method on a new thread.** Apparently it's a common misconception in developers first introduced to `async` that use of the keyword automatically spins up new threads for processing. This is not the case. Using the `await` keyword *may* spin up a new thread to continue execution of a longer running, parallel task, but that's not always a given.
- **Async doesn't always "just work".** I won't go into detail here, but there are issues like deadlocks to be aware of when creating asynchronous methods in .NET (see this older, but excellent article on the subject: http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115163.aspx). Using `async` also doesn't guarantee a boost in performance or optimal use of threads, though it generally does, and should be considered only for I/O or network-bound processes where the bottleneck is in latency not attributable to CPU processing.

Implementing Async in Web API Methods

At a high level, implementing asynchronous methods, or converting existing synchronous ones, is easy. The introduction of the `async` keyword to a method declaration in .NET tells the compiler that internally, some thread management logic will probably occur. Then

within that method, the `await` keyword instructs the runtime to free up calling threads while some background work is done on an `async`-supported task. Lastly, the return type from these asynchronous methods must be one of `Task<TResult>` (when a return type is expected), `Task` (if no return is expected), or simply `void` (used *only* to support asynchrony in event handlers, and required in this case.) Lastly, it's typical convention to append "Async" to your methods if they are declared with an `async` modifier; for example, `GetProductDetailsAsync()`, though in Web API you should use routing logic to define your endpoint names, so these conventions shouldn't be exposed in your public API.

A Simple Asynchronous Endpoint

Following is a simple Web API method for a GET request. As you can see, we are using the `await` keyword here which allows our server to free up resources while the slower network call is made.

```
[HttpGet]
public async Task<IHttpActionResult> GetCoolSpaceData()
{
    using (var client = new HttpClient())
    {
        client.BaseAddress = new Uri("https://api.nasa.gov/");
        var response = await client.GetStringAsync("planetary/apod?
api_key=apikey");
        if (!string.IsNullOrEmpty(response))
        {
            return Ok(response);
        }
    }
    return NotFound();
}
```

Seriously though kids, NASA has a free API. The future is pretty rad.

When the line containing the `await` keyword is reached, the runtime sets up a process known as continuation, which effectively copies the current context to a new background thread from the thread pool and returns control to the calling method. This means the calling thread is no longer blocked while waiting for the HTTP response. The result in Web API is a bit ambiguous, but consider how this works in a client application having a UI thread; control is returned to the sole UI thread and this background processing and network request no longer causes latency in UI rendering.

Summary

Using `async` in Web API can provide incredible performance gains with very little

overhead or change in code. Creating asynchronous methods gives us the ability to use the `await` keyword, which automatically handles the wiring necessary to complete longer-running processes on an alternate thread when possible, allowing our calling thread to continue without being blocked. File or network operations always introduce some latency which we've come to expect as developers, however in many cases we have other code not dependent on those requests which we can run independently and in parallel.

The big win in using `async` and `await` in Web API is in allowing the runtime to perform thread management automatically and return otherwise blocked threads to the thread pool where they can service other requests to our API. More threads equals more processing power and better scalability, and `async`/`await` makes for an easy improvement over a synchronous approach.

TEN: Document Your APIs Easily With Swagger

Everyone Hates (Writing) Documentation

Writing documentation is a pain. More often than not we spend the bulk of our time writing amazing features for our software only to skip or rush the documentation process that ties it all together. Moreover, creating a testing or example toolkit for our code takes even more work and is more likely to fall through the cracks. And let's not forget about updates and breaking changes, those critical times when clear documentation becomes even more important but is more likely to fall by the wayside.

Fortunately the process of documenting and providing testable examples of your APIs can be automated (you *do* automate everything you possibly can, right?) and simplified using Swagger, an awesome and free framework for interactive API documentation.

Introducing Swagger (And Swashbuckle)

Swagger is a code-agnostic platform for testing and documenting RESTful APIs. It is comprised of Swagger UI, a web-based testing tool integrated right into your API, and a standardized JSON data source known as a Swagger Specification (Swagger Spec) which defines the public interface of your API, including all requests and responses, return types,

HTTP methods, and more. Swagger UI reads the Swagger Spec from your application and outputs a beautiful interface with documentation, and forms for testing every endpoint in your API.

Now, the Swagger Spec is complex and supports a lot of different options, but fortunately for us there's a great NuGet package called Swashbuckle which handles wiring this all up for us. You literally install Swashbuckle, run your application, visit */swagger* and boom, you're presented with incredibly clear, actionable insight into your public API. What's more, if you use XML-based comments (triple-slash comments) in your code and opt to output them to an XML file, a simple configuration includes them in your output as well. The beautiful thing here is that your documentation exists right alongside your code making documentation maintenance easy (and version-controlled along with the API), and allows you to make those updates while you're already in the code and the information is fresh in your mind. It also puts the responsibility of documentation on the developer, the one with the most knowledge about the API (just make sure you code review this stuff; we developers can be pretty awful at explaining our thought process.)

Installing Swagger and Swashbuckle

Swagger and Swashbuckle can be installed via NuGet with the following command:

```
Install-Package Swashbuckle
```

Basically, that's it. Doing so will give you baseline functionality and provide a UI plus testing tools for all of your public API methods, again available at the path */swagger* on your running API; the UI is pretty self explanatory. The installation process will also add a file into your project at *App_Start/SwaggerConfig.cs*, a comment-filled file with examples

of options you can enable or configure. Of these, one example to note is that which enables XML-based comments, so if you want to take advantage of this, you'll need to specify a path to your XML file, something like this:

```
var commentsFile = Path.Combine(HttpRuntime.AppDomainAppPath, "bin",
"MyApi.XML");
c.IncludeXmlComments(commentsFile); // c is a SwaggerDocsConfig for
EnableSwagger() setup
```

Not all of the XML comments you enter will show up in Swagger, but the most important ones and those which provide the best output in Swagger's interface are `summary`, `param`, `remarks`, and `response`.

Summary

As developers we sometimes dread writing documentation and providing tools for testing our APIs. We like to think about everything in terms of code and to us, reading a header file or looking at an `ApiController` implementation is more than sufficient to call "documentation", even though we drool over well-documented APIs like those of Stripe, Twilio, or Netflix (just seeing if you're paying attention here.)

So, Swagger and Swashbuckle fall right into that happy-medium place between ease of use and usefulness. Hand-writing documentation makes for clear, well-explained information but often gets put aside as actual development takes priority. No matter how cumbersome the process is, APIs deserve clear and usable documentation which complements the application and leaves no ambiguity. We owe it to ourselves as developers to answer questions about our APIs preemptively, which is a major indicator of a well-thought-out

and smartly-architected API, and Swashbuckle helps us best do that by utilizing the code and comments we've already written, removing a majority of the grunt work.

ELEVEN: Bonus - HTTP Status Code Guide

The Only HTTP Status Codes You'll (Probably) Ever Need

According to a very, very recent Internet search I did, there are at least 50 different HTTP status codes. I'm sure you're familiar with the most common ones like *200 OK*, *404 Not Found*, *500 Internal Server Error*, and possibly not as familiar with some of the lesser-used ones like *412 Precondition Failed*, or *418 I'm a Teapot* (Google it; you know you want to.) With so many options, you may suffer from analysis paralysis just trying to decide which to use in your API.

Well, fellow developer, fear not. Most of the status codes are best suited to specific situations, many of which your API probably isn't designed to handle (unless you're doing IoT work and your server *is* actually a teapot.) Following is a list you can normally pare down to, ignoring all the others.

Common HTTP Status Codes for Web API

- **200 OK** - Used for all successful GET requests, and also usable in other methods like POST provided you're not too strict on the semantics of these codes (another code, *201 Created*, is described next and a better choice when a POST or PUT creates a resource on the server.) Use `Ok()` for this.

- **201 Created** - Used when an HTTP request creates a resource on the server, typically via POST or PUT. By definition, this should also return information about the created resource. Use `Created()` for this.
- **304 Not Modified** - Used to alert the client that the requested resource has not changed, and the client can continue using any cached version it may have. Use `Content(HttpStatusCode.NotModified, ...)` for this.
- **400 Bad Request** - Used when data is unusable for the purpose of completing the request. For example, when a required property value is empty, or missing entirely. Use `BadRequest()` for this.
- **401 Unauthorized** - Used when client can not access the requested resource *because* it's not authenticated. More succinctly, the client is not logged in. Use `Unauthorized()` for this.
- **403 Forbidden** - Used when the client *is* properly authenticated, but does not have access to the resource it is trying to access (no authorization). Use `Request.CreateErrorResponse(HttpStatusCode.Forbidden, ...)` for this (there is no built-in helper method on the controller for this status code.)
- **404 Not Found** - Used when a requested resource does not exist and possibly never has. There are other options like *410 Gone* for when a resource has been deleted, or *301 Moved Permanently* when a resource has been moved, but a 404 is valid in all cases; personally I prefer it because it doesn't give malicious users any more information than "it's not here." Use `NotFound()` for this.
- **500 Internal Server Error** - Used when your code throws an exception, or ends up

in a non-recoverable state. This typically signifies a major problem that requires intervention. Use `InternalServerError()` for this.

---

[1] https://en.wikipedia.org/wiki/Decision_fatigue

[2] http://googleresearch.blogspot.com/2009/06/speed-matters.html

[3] https://moz.com/blog/how-to-improve-your-conversion-rates-with-a-faster-website

[4] http://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/