# Welcome

## Who

Marty Mazurik, former Instructor, Microsoft Certified Partner Training & Solutions in Portland, Oregon.  
Authored 22 courses, in .NET, SQL, Exchange, C++, C#, .NET Framework and others.

Targeted 2 hours. Breakdown: Lecture 1 hour, exercises 1 hour.

## What

Knowledge Transfer: Notes and lab exericises that are links out to the source(s). This delivery provides quick-information about RESTful API development for the Dev Team at BHG to absorb and utilize as the team sees fit.

## Why

To provide an understanding of RESTful APIs: their composition and how they can be standardized, called with Postman, documented and built is important for creating uniform service layers within the organization.

## How

* I will deliver about 50-60 minutes, and try to leave time for questions using a Powerpoint slides preso to pace me, along with demo’s and code looks.
* I will have 3 exercises for the 2nd hour for you to go off and do (nothing is better than hands-on, right?)
* There will be extra credit, additionally for more advanced work and of course reading via hyperlinks and videos to further investigate the topic.
* If something is glaringly innacurate and you must interrupt, please type in the TEAMs IM chat window your question (maybe someone else will answer it). I would normally engage, throughout but want to be respectful of time, with so many developers (and some that have experience already in the topic)

## Where

I provide the solutions to the base exercises, this File, and the powerpoint in the BitBucket repo:  
<https://bhggit/projects/LM/repos/training-restapi>

Disclaimer:  
Whereas, I’m only human, this is my take on the pros (no cons) of writing a good, well patterned API with auto-doc … RESTfully.

Total team study, can help with level-setting everyone on ideas, patterns and conventions. I make no pronouncements of pragmatism or dogmatic proclamations, this is just one layout of the material based on what I see and where it may come into play using the RESTful approach.

## How Much

No charge, other than time to learn and apply your new skills.

## Goals

* Understand what REST is, and why we use it for APIs
* Know the usage of VERBS and URIs
* Learn how to design your API Endpoints and some coding conventions
* Know about auto-documentation with Swagger/ OpenAPI
* LAB Exercises: do some take-away hands-on exercises to test you skill
* Have the “Instructor Notes” document (with reference hyperlinks for further study)

## Tools

* Visual Studio 2019 or VS Code, .NET 5.0 Core
* Postman (latest)
* Swashbuckle/Swagger

# REST Overview

## REST Terminology

* [HTTP 1.1 Specification](https://tools.ietf.org/html/rfc2616) – “A great document to read” Hyper-text Protocol standard RFC 2616 © The Internet Society (1999), 176 pages that spells out the rules of the HTTP protocol in detail, these RFC docs are updates to the above spec, and make up the whole HTTP/1.1 Protocal:
* [RFC 7230](https://tools.ietf.org/html/rfc7230): Message Syntax and Routing
* [RFC 7231](https://tools.ietf.org/html/rfc7231): Semantics and Content
* [RFC 7232](https://tools.ietf.org/html/rfc7232): Conditional Requests
* [RFC 7233](https://tools.ietf.org/html/rfc7233): Range Requests
* [RFC 7234](https://tools.ietf.org/html/rfc7234): Caching
* [RFC 7235](https://tools.ietf.org/html/rfc7235): Authentication
* REST – REpresentational State Transfer it is an architectural style for hyper-media solutions presented by Roy Fielding in 2000, [his Doctoral Thesis](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). Chapter 5, on REST includes this description:  
  “REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A representation is a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.”
* JSON – JavaScript Object Notation
* URI – Uniform Resource Identifier (the pathing to the resource in your request)
* URL – in addition to a URI, tells how to access item ( http:// or https://, or ftp:// )

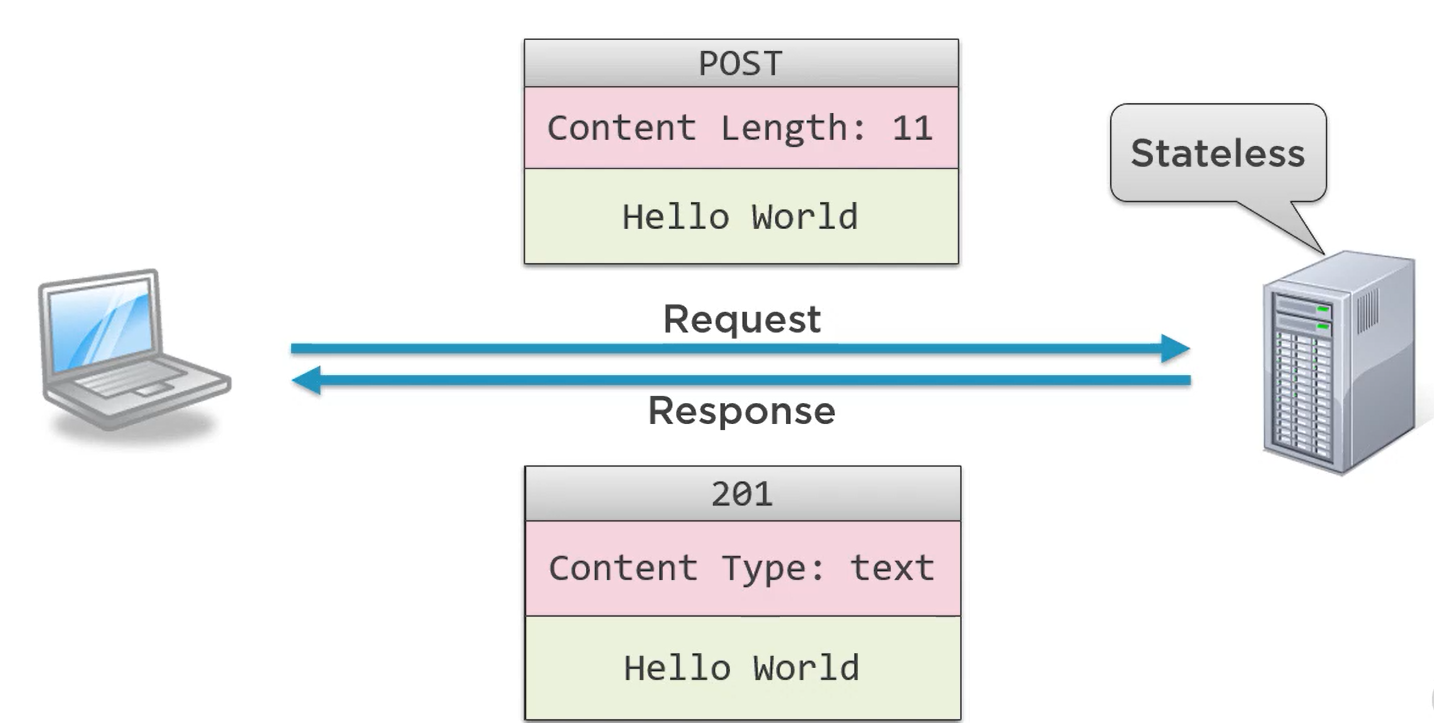
## History of APIs

* RPC’s (Remote Procedure Calls) have been around since the 70’s
* COM/DCOM and CORBA, Java RMI in the 1990s
* Web-Based APIs in the 2000’s
* XML HTTMP
* REST
* SOAP (Simple Object Access Protocal)
* GraphQL
* gRPC (and protobuf)

## Why do you need a Web API?

* Situations where a REST API may be helpful:
  + The API will be accessed by different systems
  + You are publishing the interface to customers (free or metered)
  + You are building an App that is a SPA, or mobile app
  + You are building a website (Internet or intranet)
  + You are building containerized solutions for backend services
  + You are building Cloud based solutions that other systems will access
* Some situations that *may not* require an REST API
  + You are building a traditional MVC App using WPF
  + The functionality is shareable with package libraries ( npm, nuget, etc)

## HTTP review



### The Request

### 

* Starts with a verb:

|  |  |  |
| --- | --- | --- |
| HTTP Verb | Used for | Possible return codes |
| POST | create | 201 created, 409 conflict |
| PUT | Update (replace) | 200 ok, 404 not found |
| GET | read | 200 ok, 404 not found |
| DELETE | delete | 200 ok, 404 not found |
| PATCH (many just use PUT) | Update (modify) | 405 method not allowd |
| HEAD | read (xfer status & header only) | 200 ok, 404 not found |
| OPTIONS | Describe comm options for target | 200 ok, 400 bad request |

* Usually contains **Headers**
  + - **Content Type**:  **application/json** (or text, XML, JavaScript, …)
    - **Authorization**: **No Auth** (or Bearer Token: *{followed by a OAuth 2.0 token}* )
    - **Accept**: **\*/\***  (return any content type, or could be: application/json)
    - many others possible, complete list here on [w3.or website](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html)
* (optionally) content
  + - Common to use a JSON body for request content. *Body* (short for [HTTP message body](https://en.wikipedia.org/wiki/HTTP_message_body)) is the common term used when describing the payload.
    - Example JSON Body:

{

"Temperature": 49.5,

"Timestamp": "12/31/2020 11:00 AM",

"City": "Seattle",

"State": "WA",

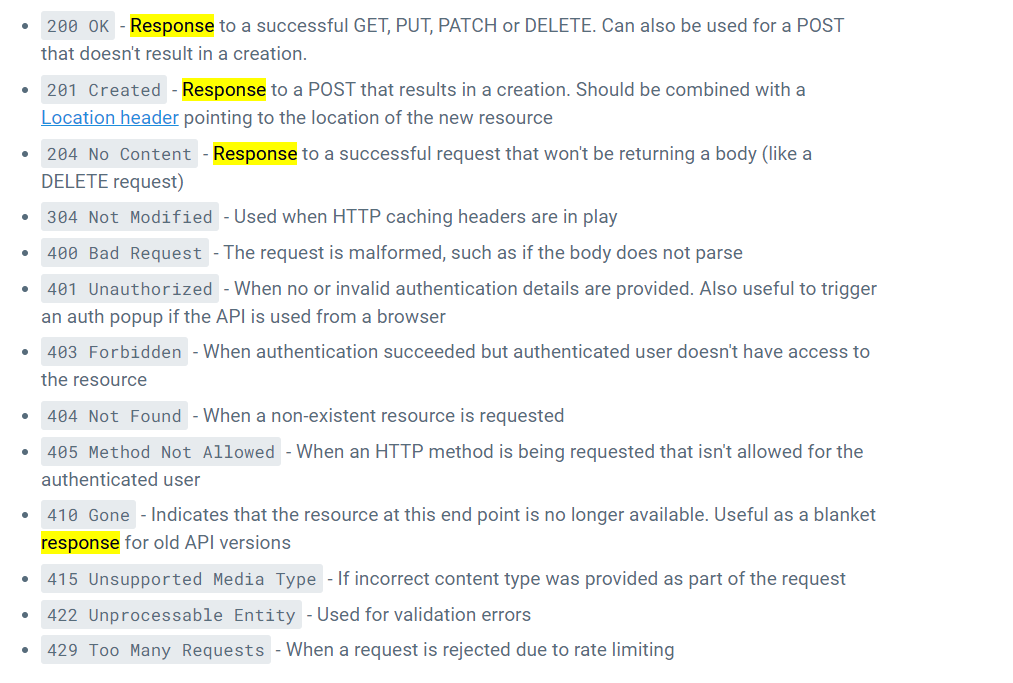
"Zip": "98101"

}

## Response

I won’t talk about responses, too much, other than to say. **Stay terse, my friends.**  
Response == CODES (only), unless a GET, then a Body-JSON.

Resist passing back exception information, bubbled up, etc to the caller. It should all be trapped in the controller to pass back the HTTP codes. The exception is HTTP 201, where not a Body, but a header called “Content-Location” can be passed back as a URI. See [RFC2616, section 14.14](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html).



## What is REST ?

REST is an architecture. An architecture for *interfacing*. Interfaces are the methods of interaction and the root word in Application Programming Interface, or *API*.

In order *to be considered RESTful*, you have to abide by six guiding principles.

### Six guiding principles

1. Client-server – separate the UI from data storage concerns
2. Stateless – the request must contain all the information needed to understand the request. No stored context.
3. Cacheable – data within a response is: implicitly or explicity labeled as cacheable or non-cacheable.
4. Uniform – defined by four interface constraints
   1. Identify the resources (via a URI )
   2. Manipulation of said resources ( again, via URIs ) // think C.R.U.D. manipulation
   3. Self-descriptive messages ( stateless & self-contained )
   4. Hypermedia as the engine of application state, ([HATEOAS](https://en.wikipedia.org/wiki/HATEOAS)),  
      Actual meaning: A REST client needs little to no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia.  
      Know the Verbs, URI, headers and the body of the message and you are set to interact.

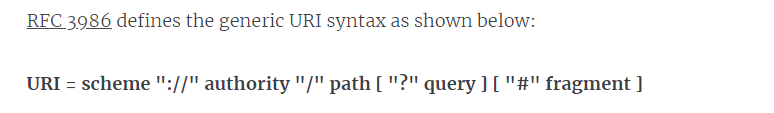
This “Uniform Interface” concept was put forth by [Roy Fielding in his doctoral dissertation](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) here, where he opins:  
*"software design on the scale of decades: every detail is intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency. Unfortunately, people are fairly good at short-term design, and usually awful at long-term design".*

Put another way: “constructing uniform API URIs is how to achieve great, long-term design”  
Shortened: Design uniform URIs for your RESTful API

1. Layered Architecture – [see here #5](https://medium.com/future-vision/the-principles-of-rest-6b00deac91b3)
2. Code-on-Demand (optional)

# Designing a RESTful APIs

Whether in your code, creating a path to a controller method, or in POSTMan a tool for debugging REST APIs you will be working with URIs.



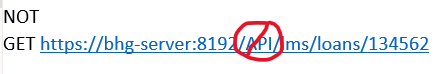
## Rules for URI Design

For the most part you should use the “Collection Pattern”, and think C.R.U.D.

[From 8 Rules for Rest API URI Design](https://blog.restcase.com/7-rules-for-rest-api-uri-design/), we get these principles:

1. No trailing forward slash (/) needed  
   <https://localhost:44378/weatherforecast> not  
   <https://localhost:44378/weatherforecast/>
2. Forward slash used to indicate a hierarchical relationship

<https://bhg-server:8192/loans/134562>

1. Avoid use of API to start your URI for services:  
   GET <https://bhg-server:8192/lms/loans/134562>   
     
   
2. If you have a service with multiple controllers, THAT will be the first segment of the URI

GET <https://bhg-server:8192/admin/ping> ( generic service admin controller )  
GET <https://bhg-server:8192/lms/loans/134562> ( lms service main controller )

1. Services will be know by their ( IP address or Domain Name AND Port number )
2. Use hyphens in long path segments ( a GET from blogger.com blog title)  
   <http://www.martymazurik.com/2016/12/black-bean-quinoa-kale-corn-burgers.html>
3. Underscores should not be used, use hyphens.
4. Lowercase letters should be preferred in URI paths. Forget Hungarian, or camel-casing notations.
5. File extensions should not be used, instead relying on the header’s MediaType.  
   <http://api.college.com/students/3248234/courses/2005/fall>  
   NOT  
   <http://api.college.com/students/3248234/courses/2005/fall.json>
6. Use plural form of the noun  
   <https://bhg-server:8192/loans/134562>
7. Use appropriate status code in response

## Exceptions to rules (Edge Cases)

### **What about actions that don't fit into the world of CRUD operations?**

This is where things can get fuzzy. There are a number of approaches:

* Restructure the action to appear like a field of a resource. This works if the action doesn't take parameters. For example an activate action could be mapped to a boolean activated field and updated via a PATCH to the resource.
* Treat it like a sub-resource with RESTful principles. For example, GitHub's API lets you [star a gist](http://developer.github.com/v3/gists/#star-a-gist) with PUT /gists/:id/star and [unstar](http://developer.github.com/v3/gists/#unstar-a-gist) with DELETE /gists/:id/star.
* Sometimes you really have no way to map the action to a sensible RESTful structure. For example, a multi-resource search doesn't really make sense to be applied to a specific resource's endpoint. In this case, /search would make the most sense even though it isn't a resource. This is OK - just do what's right from the perspective of the API consumer and make sure it's documented clearly to avoid confusion.

### Another perspective Example courtesy of Kenneth Lange, article [“Don’t Limit your REST API to CRUD Operations”](https://www.kennethlange.com/dont-limit-your-rest-api-to-crud-operations/)

/rockets/43/launch-rocket

When breaking with the Collection Pattern I really like the verb-noun naming of the URI, such as **launch-rocket**

### Further viewing (perspective) on Domain Driven Design with REST Additionally courtesy of Jim Weber, [in this talk](https://www.youtube.com/watch?v=aQVSzMV8DWc) ( 1 hour video preso on YouTube)

# Return Codes

It is very important that as a RESTful API, you make use of the proper HTTP Status Codes.

The mostly used status codes:

200 – OK

Everything is working

201 – CREATED

A new resource has been created

204 – NO CONTENT

The resource was successfully deleted, no response body

304 – NOT MODIFIED

The date returned is cached data (data has not changed)

400 – BAD REQUEST

The request was invalid or cannot be served. The exact error should be explained in the error payload. E.g. „The JSON is not valid “.

401 – UNATHORIZED

The request requires user authentication.

403 – FORBIDDEN

The server understood the request but is refusing it or the access is not allowed.

404 – NOT FOUND

There is no resource behind the URI.

500 – INTERNAL SERVER ERROR

API developers should avoid this error. If an error occurs in the global catch block, the stack trace should be logged and not returned as a response.

# Versioning

Much of the material glommed onto from <https://restfulapi.net/versioning/>

“In the real world, an API is never goint to be completely stable.”

## When to version

Once an API is completely published for consumers, you have to manage versioning.

* A change in the request or response type(s) of an API call
* A change in the response data for one, or more calls
* Removing any part of the API

***Breaking changes*** should always result in a change to the major version number for an API

### Avoid versioning “churn”

Consumers and producers of RESTful APIs should understand that during development cycles an API will go through numerous changes, sometimes *multiple times a day*.

Providing auto-documentation is one way to help communicate with consumers while in the development cycles.

Full releases of an API should correspond with major Product releases. That is when to fixate the version number, and be ready to increase the version for the next major release with changes that are required.

## How to Version

REST doesn’t impose or suggest guidelines about versioning.

Two methodologies:

1. URI Versioning  
   <http://myapi.bhg-inc.com/v1>
2. Custom Request Header  
   Accept-version: v1  
   Accept-version: v2

# Auto-documentation

There is a whole separate course on Swagger on Pluralsight:  
<https://app.pluralsight.com/library/courses/getting-started-swagger-tools/table-of-contents>

## What is Swagger

Swagger is a auto documentation feature that creates HTML pages that describe REST APIs so humans can understand the capabilities without accessing the source code directly.

Swagger was developed by SmartBear, and donated to the OpenAPI initiative in 2015, and both names are used interchangeably. But note distinction:

* OpenAPI is a standard specification
* Swagger is tooling that implements the standard

## Why Swagger

* Uses common language devs and non-devs can understand
* Can be used for testing and bug fixing (invoking methods)
* Save writing (and editing) documentation on API – and keeping it in sync (yuk)
* Industry standard, donated to OpenAPI by SmartBear in 2015, and still maintained by SmartBear

## How it works

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-5.0>

* You register middleware in your .NET cor Startup.cs
* Detection by SwashBuckle generator creates an openapi.json file automatically from your POST, GET and other methods in code (by the compiler Attributes)
  + Also picks up the Body JSON types that will be received by the REQUEST methods and formats them for documented types !
* There is a SwaggerUI that then will display in HTML the openapi.json file that is generated
* Additional provisions within Swagger markup for:
  + Method comments
  + Version info can be added,
  + Return type’s JSON

## Implementing Swagger in .NET Core READ (also in your Exercise 3 hands-on)

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-5.0&tabs=visual-studio>

# Exercises

## Exercise #1 – Implementing Simple POST (Create) for MyAPI

1. Skip to the end of the document and do steps, Section-9  
   “Create a RESTful WebAPI Project in Visual Studio 2019“ and create a project called “MyAPI”
2. Make sure you have Postman installed to test your API, [more here](https://www.postman.com/downloads/)
3. Create a simple Data element class, visit this site, [Json2Csharp.com](https://json2csharp.com/)  
   and type in (or cut-n-paste) this JSON, it is a website that will build your C# entity class out of the JSON (if the JSON is correct). Press the green, Convert button.

{

"FahrenheitTemperature": 78.8,

"Timestamp": "12/31/2020 11:00 AM",

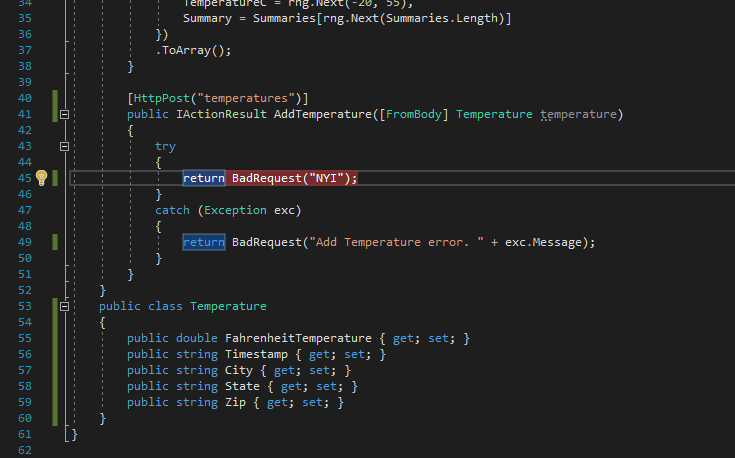
"City" : "Seattle",

"State" : "WA",

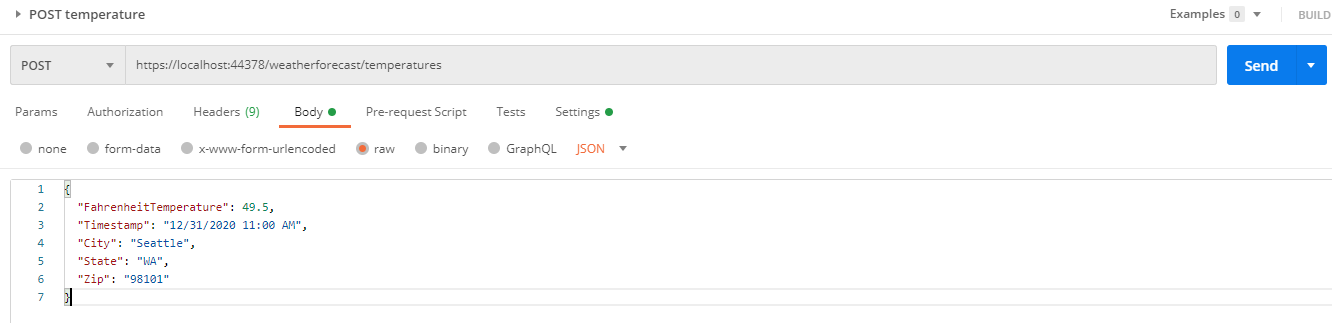
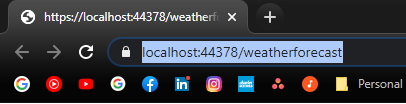
"Zip" : "98101"

}

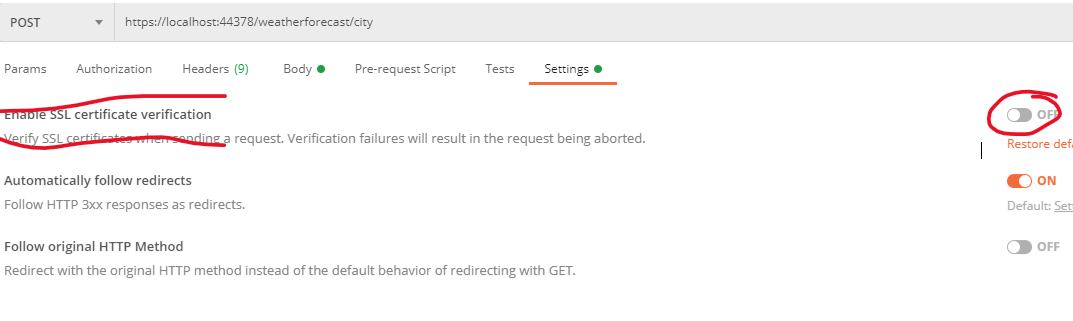
1. Copy the C# class into your project’s controller (note: this is not best practice, just a quick lab), change the name of the class from “Root” to **“Temperature”**
2. Add a POST verb for “Temperature to the WeatherForecastController.cs file, with the following code and compiler attribute [HttpPost( ) ]



This is a nice thing to wire up the controller, and not have to worry (at this time) about anything but the REST design and the passing of the Body of the Request (in this case the Temperature entity).

1. In Postman,   
   Create a new Collection (“MyAPI”) and a new Request ( POST ) as below
   1. To see your URL domain (localhost) and : PORT # for your VS2019 running project, look in the browser window that opens when you run your project:  
      
2. Make sure your MyAPI, Visual Studio IISExpress REST API code is running, and set a breakpoint inside the try/catch block

NOTE: recently the HTTPS Certificate in Visual Studio has been barking, if this happens, POSTMan has a way to shut off this warning message under “Settings” for the Request



How do we know its working?

1. You should stop at the breakpoint in your code on the return statement.
   1. check with QuickWatch your local variable (Temperature) see that it was serialized into your entity correctly.
2. You should get a 400 – Bad Request, with additional text: “NYI” ( for not yet implemented) response back in POSTman.
3. Extra-Extra credit (advanced):  
   ActionResult returns are convenience objects. They have a Bad Request (400) but not a Not Implemented (501).  
   Research how you would modify code to return an HTTP 501, using ActionResult (or not), to be technically accurate on the 501 return code for NYI.

## Exercise #2 – Implementing More ‘Collection Pattern’ Methods for MyAPI

**Assumption:** Zip+Timestamp represent a unique entry in the system. You cannot store in the system more than one entry with the same Zip/Timestamp combination. Think of it like a multi-key database constraint.  
  
a) Create the REST controller code for a method that will “Update” an existing Temperature record.

What VERB is involved in this update?  
What URI

b) Create the REST controller code for a method that would DELETE all Temperatures

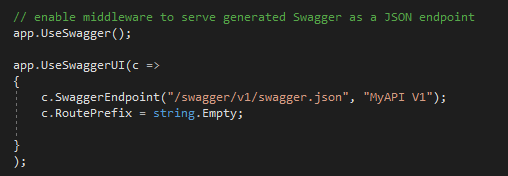
c) Create the REST controller code for a method that would DELETE all Temperatures for a Specific Zip

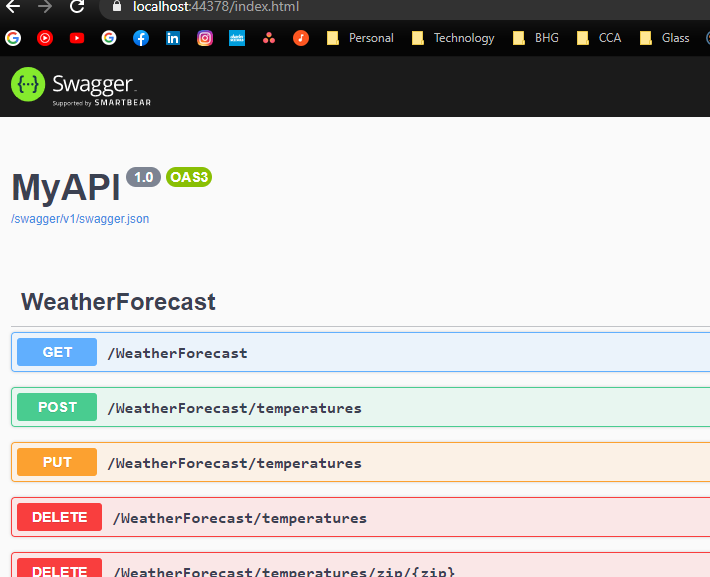
d) Create the REST controller code for a method that would DELETE a single Temperature by Timestamp and Zip

## Exercise #3 – Adding Swagger to MyAPI

Solution:

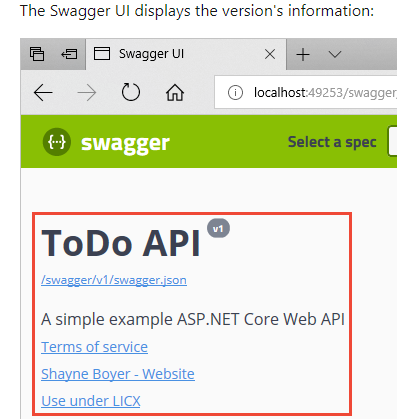
<https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-5.0&tabs=visual-studio>

1. Install Swashbuckle.AspNetCore package
2. In Startup.cs file ConfigureServices method, add: (after services.AddController)  
   
3. In the Configure method, add: (before the app.UseRouting(); method call)  
   
4. Run your application in VS2019
   * From a browser, see the Swagger generated JSON file: <https://localhost:44378/swagger/v1/swagger.json>
   * With c.RoutPrefix set = string.Empty we can get to the SwaggerUI by simply hitting the service endpoint
   * <https://localhost:44378/index.html>



### Extra Credit

Following the instructions down to <https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-5.0&tabs=visual-studio>  
The parts where you can:

1. Add API description information to be displayed, and test, as in:  
     
   
2. Add code to allow triple-slash comment descriptions to be displayed, and test, as in:  
   

### Extra-Extra Credit (advanced)

1. Add code from the above mentioned document where you can get a Response Type returned from a POST from you MyAPI documented in Swagger, as in:

[HttpPost]

[ProducesResponseType(StatusCodes.Status201Created)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

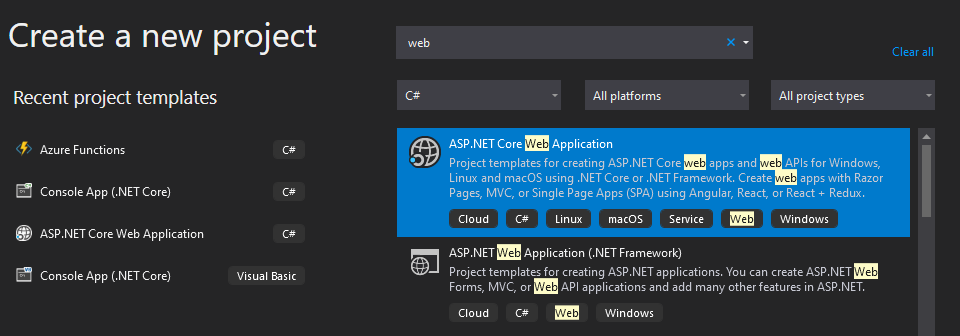
# Advanced study

Paging, Sorting and Data shaping course on Pluralight:

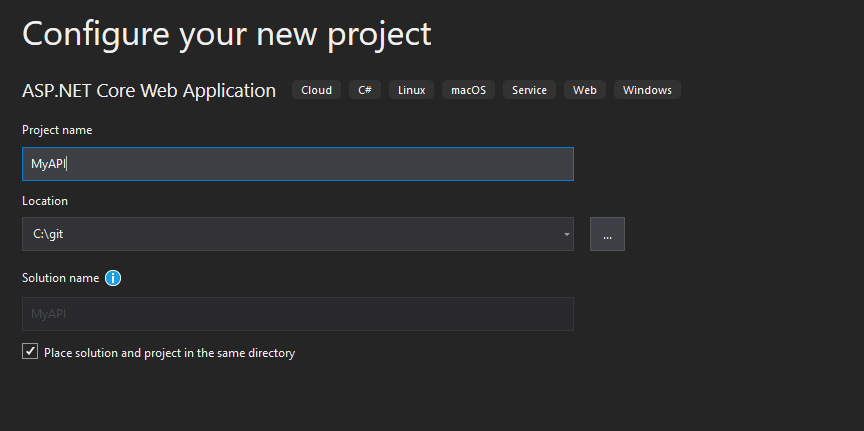
<https://app.pluralsight.com/library/courses/asp-dot-net-core-3-advanced-restful-concerns/table-of-contents>

# Create a RESTful WebAPI Project in Visual Studio 2019

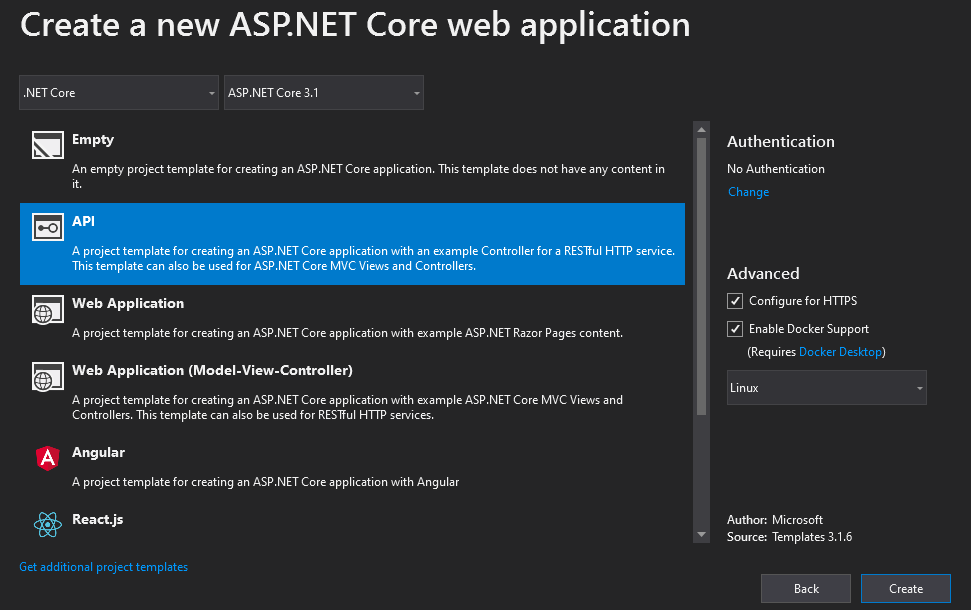
1. Create a new project, ASP.NET Core Web Application



1. Project Name: MyAPI

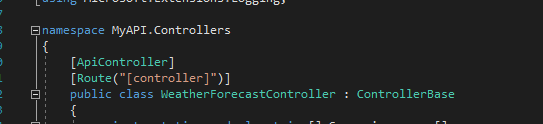


1. Defaults to ASP.NET Core web application called “API” already, so great!



For now we’ll leav the Enable Docker Support checked, if we decide to go down the container route in the future, the Dockerfile will be auto-generated. Because it is .NET Core 3, we can leave Linux as the default docker image target O.S. Targeting the Linux kernel makes the most sense for portability, and hosting options.

1. Checking out the RESTFul API automatically created by Visual Studio
   1. Under  you will see WeatherForecastController.cs VS2019 wizard created this sample controller file for us. Normally, we would delete it or copy it to a name pertinent to our project need. For now let’s examine it:
      1. [] brackets surround C# compiler *attributes*:
         1. [ApiController] says this class is a controller, so register it as such at App startup in a feature explained here: [*Endpoint routing*](https://aregcode.com/blog/2019/dotnetcore-understanding-aspnet-endpoint-routing/), introduced in .NET Core 3.0 and performed by the Startup.cs file’s configure method endpoints.MapControllers();
         2. [Route(“[controller]”)] is really nice. It says the name of the source file (WeatherForecastController) will be parsed prior to ‘Controller’ and the API can use the term ‘WeatherForecast’ in the URL and it will route here, to this particular controller.

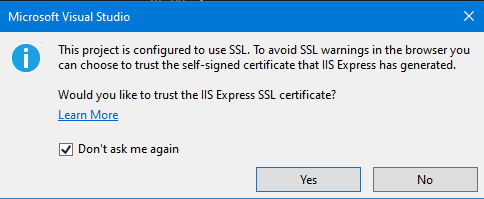


Example  
  
GET http://localhost/weatherforecast/

* + 1. [HttpGet] attribute  
       putting this inside our WeatherForecastController class, above the   
         
       [HttpGet]  
       public IEnumerable<WeatherForecast> Get()

signifies that for the GET REST call above, this is the method to call in the controller.

* 1. In your toolbar, in VS2019, make sure IIS Express is visible 
  2. F5 or Run (press the green arrow)
     1. If this message comes up



Select Don’t ask again, and click “YES”, and “YES”

* 1. <https://localhost:44378/weatherforecast> performs an HTTP GET and displays some JSON return data. You’re method was called.