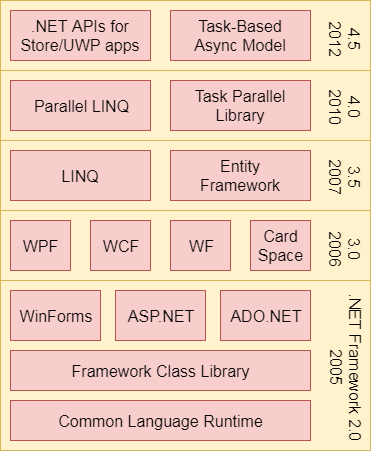
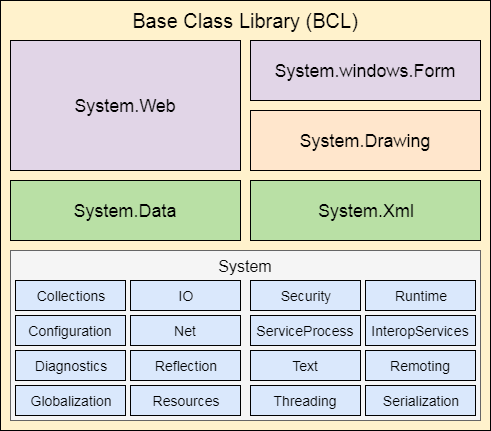
.Net Framework

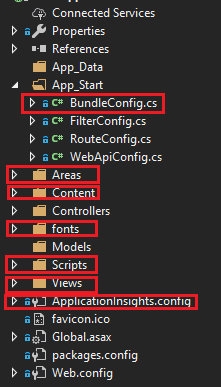
* It is used to develop applications for web, Windows, phone.
* It contains multiple class libraries known as Framework Class Library (FCL).
* These program need a Common Language Runtime to Run.
* This framework provides various services like memory management, networking, security, memory management, and type-safety.
* The .Net Framework supports more than 60 programming languages such as C#, F#, VB.NET, J#, VC++, JScript.NET, APL, COBOL, Perl, Oberon, ML, Pascal, Eiffel, Smalltalk, Python, Cobra, ADA, etc.
* **Four main components of .Net Framework** 
  + **Common Language Runtime**
  + **Framework Class Library**
  + **Core Language (Winforms, ASP.NET, & ADO.NET)**
  + **Other Modules ( WCF, WPF, LINQ, EF, Parallel LINQ, TPL)**





# Creating a WEB API

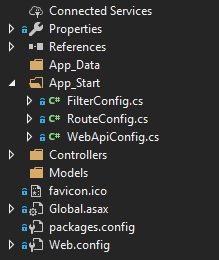
We have to create a simple web API framework application. Initially the app has lot of boiler plate code. Since in this project we are just building the web services we don’t need any UI code so we can remove the following folders.



In the **Global.asax** we need to remove the following two lines

* using System.Web.Optimization;
* BundleConfig.RegisterBundles(BundleTable.Bundles) : Removing the bundle registration part since this is not needed for the web api project.
* We will also remove some of the NuGet packages: Microsoft.ApplicationInsights related packages, Modernizr, WebGrease, Antlr, bootstrap, jQuery.

The following diagrams shows the clean project structure.



# Installing Project Dependencies

Install-Package EntityFramework -Version 6.2.0

Install-Package Microsoft.AspNet.WebApi.Cors -Version 5.2.6

# Entity Framework

Entity Framework is a mature ORM build by Microsoft.

Let’s now set up our database connection. We will add a connection string to Web.Config. Make sure to add it inside of <configuration>, after <configSections></configSections>

# Enabling CORS

Since the client will be running on a different port and address so we need to configure CORS middleware to the pipeline.

In the **APP\_Start** Folder the following needs to be added to Register method

We use statemement **using System.Web.Http.Cors;**

And add following statement to the Register method of WebApiConfig Class

*// Enable CORS for the Angular App*

var cors = new EnableCorsAttribute("http://localhost:4200", "\*", "\*");

config.EnableCors(cors);

# Setting default formatter for Asp.Net Web App

It uses default XML formatter. We have to remove it and set JSON formatter as the default formatter.

The following code needs to be added in the end of the Register Method.

*/ Set JSON formatter as default one and remove XmlFormatter*

var jsonFormatter = config.Formatters.JsonFormatter;

jsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

config.Formatters.Remove(config.Formatters.XmlFormatter);

jsonFormatter.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Utc;

For this we will need Newtonsoft Json Package

# Routing Table

By default, the server uses the Routing Table located in App\_Start/WebApiConfig.cs.

# Adding Swagger to .NET Framework Web App

* Installing Swahscbuckle Nuget Packag e



* After installing this nuget package in App\_Start folder a new file named SwaggerConfig would have been created automatically.
* Now we need to configure swagger by adding following lines to enable Swagger and SwaggerUI

GlobalConfiguration.Configuration

.EnableSwagger(c => c.SingleApiVersion("v1", "A title for your API"))

.EnableSwaggerUi();

# Routing

# Filters

# Request Processing Pipeline

Configuration Management in .NET Framework

# Course - .NET Framework

Web API is powerful framework to build REST Services.

* **Web API –**
  + It is implementation of MVC design pattern in ASP.NET
  + It is framework for building Web APIs i.e. Http Based services on top of .net framework.
  + Allow easy implementation of Http verbs and URL Patterns to specific Controller classes and actions class methods using Route definitions.
  + The makes is ideal for making REST serves as they also work on http verbs
  + The .net web api is lightweight and designed for extensibility and modification .
  + It includes the Authentication and Authorization models.
  + Out of box JSON and XML support connect negotiation.
  + Web API is not based on WCF
  + Web API services are not SOAP-based and the framework has no out of the box support for SOAP services or the WS-\* standards.
  + Web API doesn’t support a lot of out-side of the http.
  + WCF can also be used for creating RESTful services but a lot of configuration is required to do so.
  + Web Services are used to implement services which are independent of transport/protocol independent.

## Terminologies

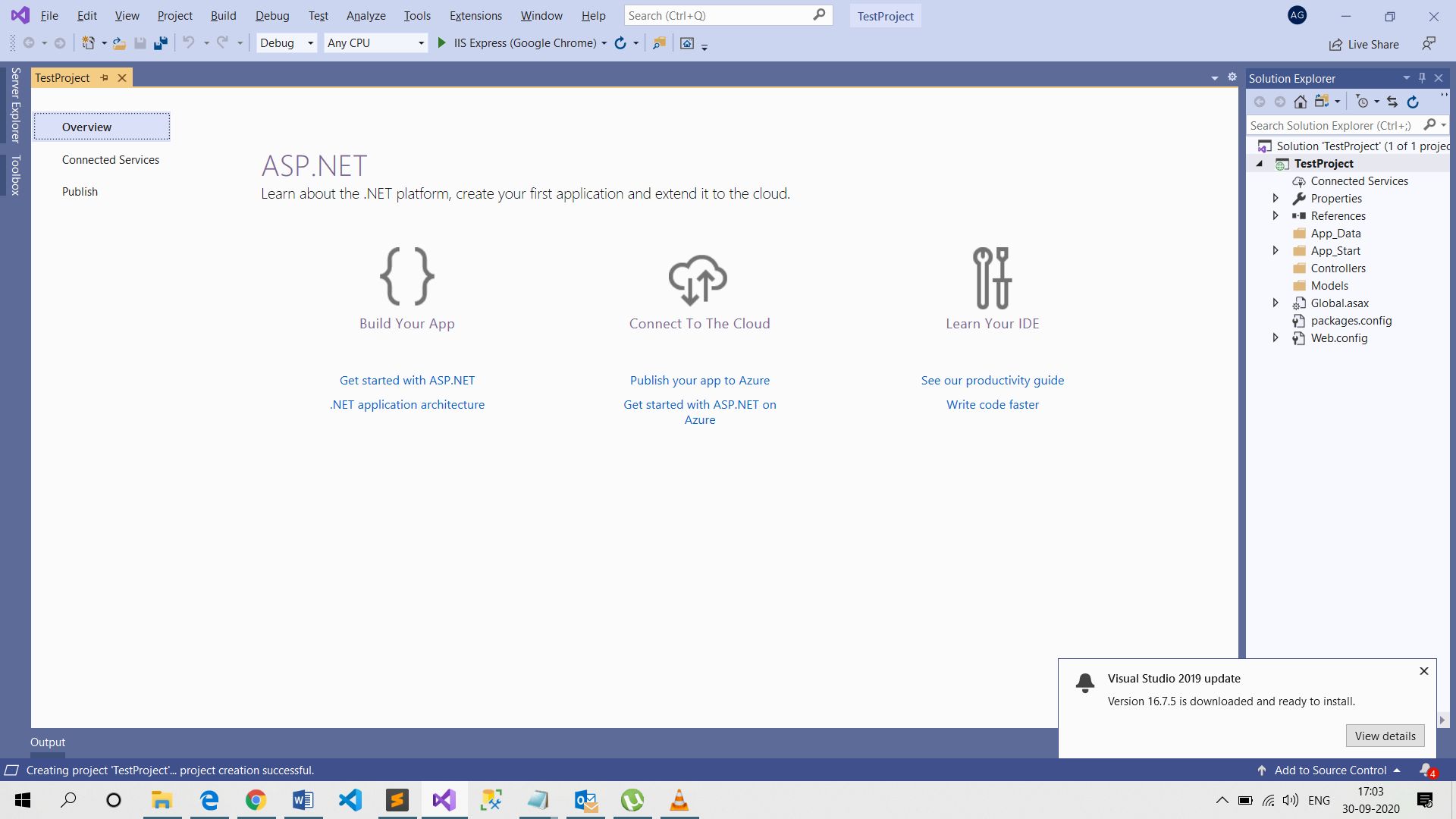
* **Http verbs –** It is standard http methods like GET, POST, PUT, DELETE, PATCH.
* **Controller –** An object class derived from ApiController base class that implements the logic of a specific web service call.
* **Action Method –** A class method of a Controller class that implements the logic of a specific web service call.
* **Request Verb** – This verb describes what needs to be done with the resource.
* **Route –** A mapping of http verb and a URL pattern to a particular Controller and Action method. The routes defines the Client Callable Apis
* **Pipeline –** the chain of built in and custom message handlers and filters that the HTTP request and response ”flow” through to reach and return from your Controller Action Methods.
* **The pipeline is where you extend the behavior of your services.**
  + Custom Http Headers
  + Authentication and Authorization Tokens
  + Metrics Logging
  + Content negotiation
  + Basically everything outside business logic of Action methods.

## How to use WEB API?

* Create a controller class with action methods that implement your web service business logic
* Define routes to map particular HTTP verbs and URL Pattern to these Controller Action Methods.
* When necessary, create extensions (called handler and filters) to support any client integration customization required. For example: to support custom http headers, authentication token, data formats, metrics, logging, etc.

# Creating a web API app in .NET Framework

* You will be tempted to use the Web API template but this is for a full blown ASP.NET app with MVC or UI so select Empty and then on the right side check box the Web API.



* In global.asax this is where the start is configured.
* In App\_Start we have WebApiConfig class which is very important and this is where all the registrations do take place of custom handler, filter, route mapping takes place.
* This line “**config.MapHttpAttributeRoutes();”** maps the route which were defined through the attribute routing.
* Below this there is a template style route so this is mapped by the template.
* Now add a controller and select the type of controller as WEB API 2 Controller with Empty or Read/Write actions.
* The name of a controller has to end with Controller. These controller derives from ApiController.
* The default result we will get in XML as browser will do content negotiation with the API and API will say it will return in XML. We can specify what type of content we want.
* In rest world when we create a resource we should return the Http Status Code Created.
* **New Project Template** 
  + In a .net web app there is a template Azure API template. For this you will need Azure SDK Installed in your visual Studio.
  + This will have default Swagger implemented which is like used for documentation and API testing.
  + This will have API action method with extra swagger attributes.
  + There is additional references for Application Insights, also Identity model jwt .
  + There is default application Insights configuration file. Although this project is a Azure template but we can run it on-prem and send the telemetry data to azure.
* If you choose the **Web API** template while creating project the controller folder will have both the web API as well as the MVC controllers. The MVC controller inherits from Controller class present in System.Web.Mvc.

# Tools for integration testing

* Browsers are not ideal mechanisms to test our Web services.
* **Fiddler –**
* **Postman**
* **Using Swagger via Swashbuckle –** This will add a Swagger Configuration file.

# Routing

## Template Style Routing

* The template routing model has series of rules to figure out which action method is right one for an URL.
* EG – Get() -> GetAllProducts() will work as same as the verb is in the name and method has no parameter. This helps us gets more control on the method name.
* Eg you can have a method name CreateProduct but annotate it with [HttpPost] to tell what verb the method is instead of having the verb in the name.
* We can add our own route template in WebApiConfig Register method.

config.Routes.MapHttpRoute(

name: "LegacyAPIDefination",

routeTemplate: "api/prod/{id}",

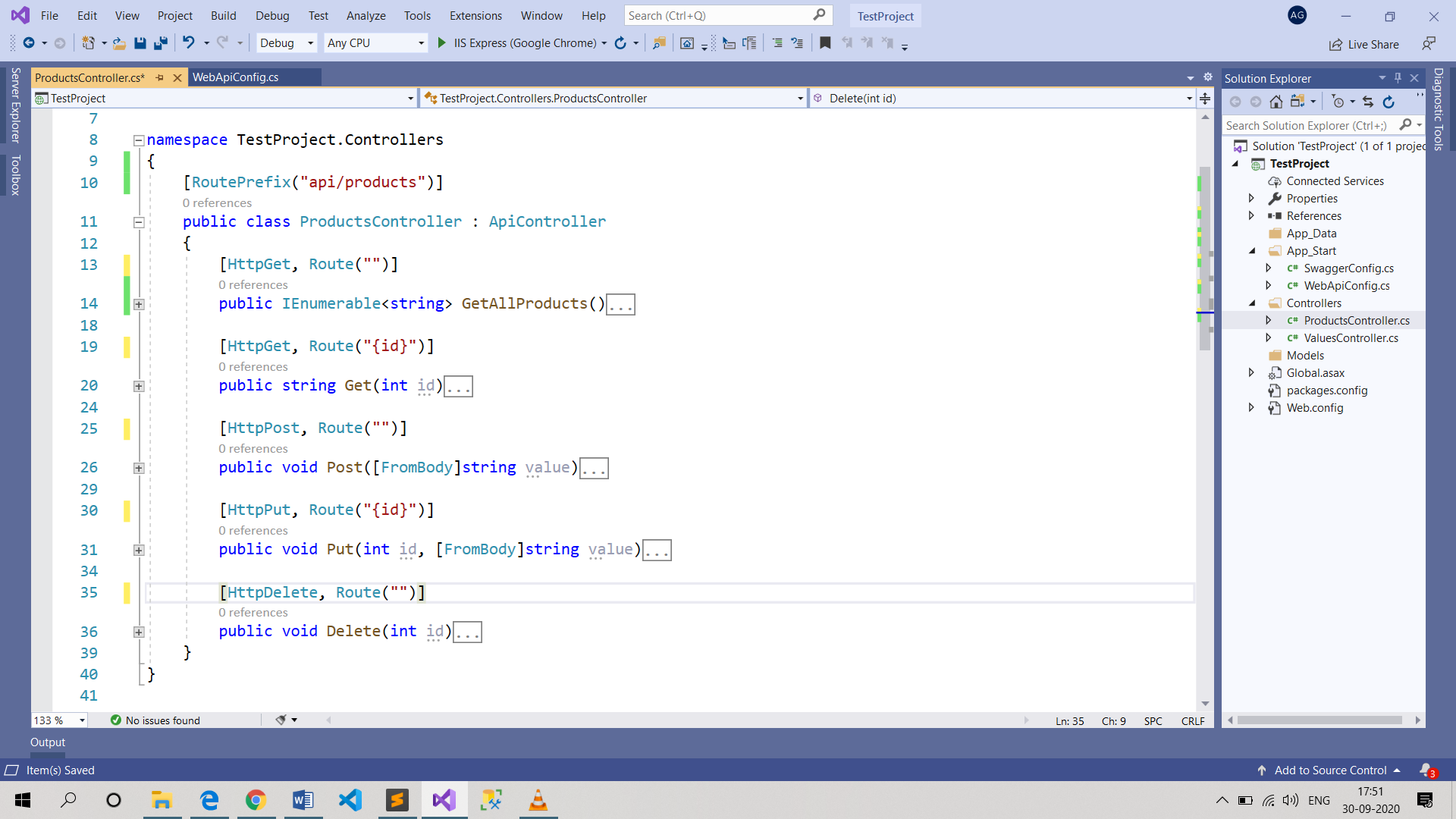
defaults: new {contrtoller = "products", id = RouteParameter.Optional }

);

## Attribute Based Routing

* This was introduced in Web API 2.0
* You can define the route prefix at a controller level like [RoutePrefix(“api/Products”)]
* And post this we can define attribute routing for all the action methods. [HttpGet, Route(“”)]

More examples [HttpGet, Route(“{id}”)] this id name should match with the param name



* In attribute routing style we have broken the link between name of method and the route like you don’t need to have Get in the name of method.

## Built in Route Constraints

* When we have route parameters we haven’t restricted to type. Like if we say we want int id , there is no way of restricting in URL that only.
* We can perform route param validation in route definition itself.

[HttpGet, Route("{id:int}")]

* We can also use multiple constraints like [HttpGet, Route("{id:int:range(1000, 3000)}")]
* If you put a validation on one route then it can go to a matching one. Like if you have a get/id and put a constraint on it but you have a put/id then it can go to put, so you need to duplicate the constraints if routes are similar.
* **The detailed documentation on the route constraints and what contraints are supported can be found on** [**https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2#route-constraints**](https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2#route-constraints)

## Custom Route Constraints

* We can built our own constraints as all of built in constraints implement IHttpRouteContraint interface.
* For this we will create an enumeration constraint to check if the passed is an enum or not.

public class EnumerationConstraint : IHttpRouteConstraint {

/// <summary>

/// Holds the type of the Enum class to validate against

/// </summary>

public readonly Type EnumType;

/// <summary>

/// Constructor taking a namespace-qualified type name of the Enum type to use

/// </summary>

public EnumerationConstraint(string type)

{

var t = GetType(type);

if (t == null || !t.IsEnum)

throw new ArgumentException("Argument is not an Enum type", "type");

EnumType = t;

}

/// <summary>

/// Internal method to convert the string enum type name into a Type instance

/// by checking all of the currently loaded assemblies

/// </summary>

private static Type GetType(string typeName)

{

var type = Type.GetType(typeName);

if (type != null) return type;

foreach (var a in AppDomain.CurrentDomain.GetAssemblies())

{

type = a.GetType(typeName);

if (type != null)

return type;

}

return null;

}

/// <summary>

/// IHttpRouteConstraint.Match implementation to validate a parameter against

/// the Enum members. String comparison is NOT case-sensitive.

/// </summary>

public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName,

IDictionary<string, object> values, HttpRouteDirection routeDirection)

{

object value;

if (values.TryGetValue(parameterName, out value) && value != null)

{

var stringVal = value as string;

if (!String.IsNullOrEmpty(stringVal))

{

// see if we can find the string in the enumeration members

stringVal = stringVal.ToLower();

if (null != EnumType.GetEnumNames().FirstOrDefault(a => a.ToLower().Equals(stringVal)))

{

return true;

}

}

}

return false;

}

}

Registering the Constraint

You will need to register the constraint in your WebApiConfig.cs file:

var constraintResolver = new DefaultInlineConstraintResolver();

constraintResolver.ConstraintMap.Add("enum", typeof(EnumerationConstraint));

config.MapHttpAttributeRoutes(constraintResolver);

Using the Constraint

In the Route attribute on your ApiController action method, use it like any other parameter constraint. The constraint requires one string parameter, the fully namespace-qualified type of the Enum you want to validate against.

If the Enum is embedded in another class use the namespace-qualified class name of the parent object "+" the Enum name (ex. My.NameSpace.MyParentClass+ColorsEnum):

[HttpGet, Route("colors/{color:enum(My.Namespace.ColorsEnum)}")]

public string GetColor(string color)

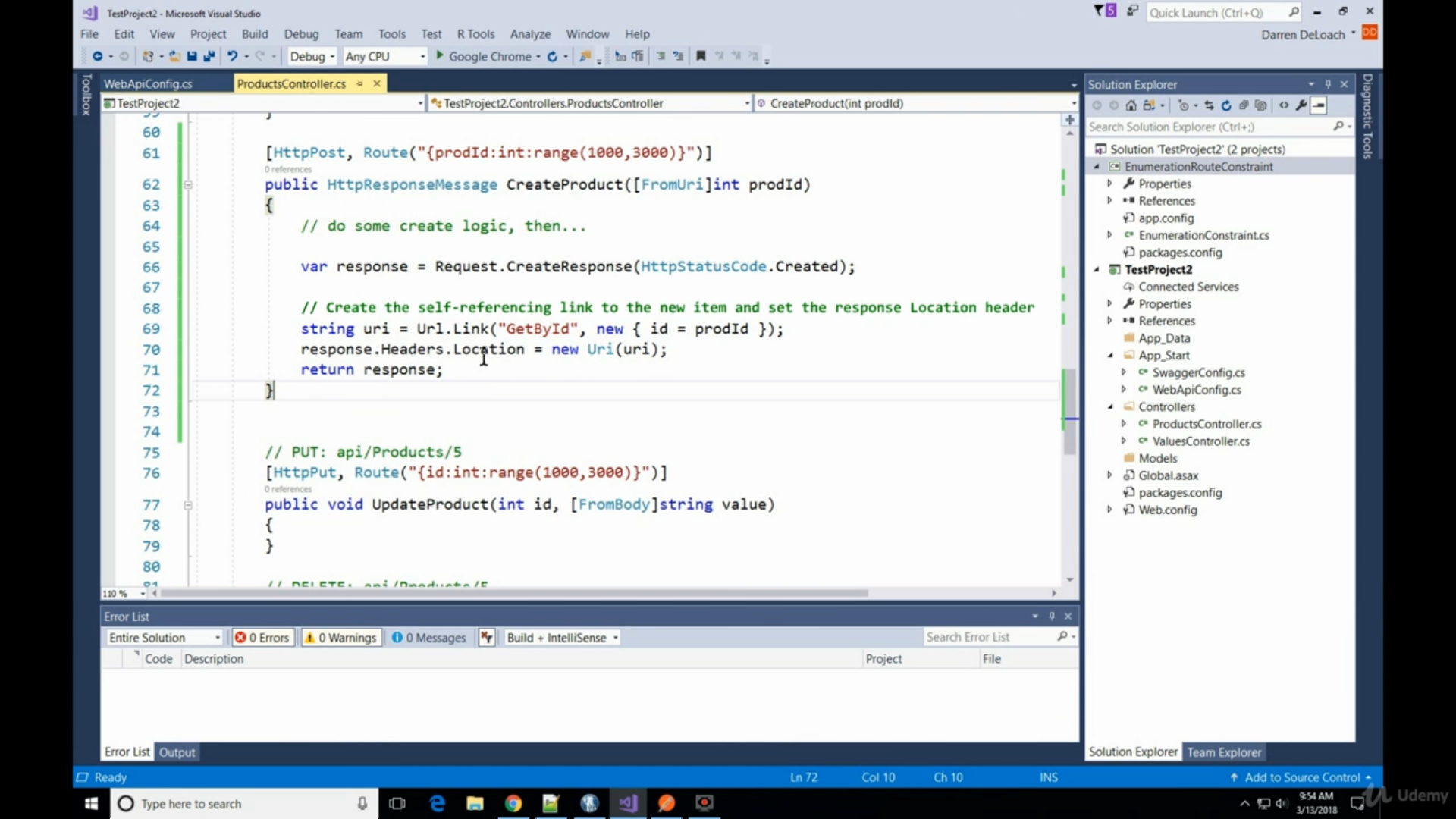
{

return color;

}

The constrained parameter is validated against the list of member names in the enumeration, where the compare is NOT case-sensitive.

## Lecture 3 Routing

* There are a lot of http verb apart from the traditional 5.
* There is an attribute [AcceptVerbs(“GET”, “VIEW”)] -> this means you want to say that either for get or view this action method will trigger.
* List of all Http Verb <https://stackoverflow.com/a/41411515>
* Apart from this we define our own verb
* We can have more than one route to the action method which means you can define more than one route to the action method. If you define route with the ~ [Route(“~something”)] then this route will not be added to the route prefix route.
* In template style routing we used to have a name for our route , this can be done in attribute routing, So we do this to achieve self-referencing APIs. For example when we have created a Resource in post call and we want to return the location of newly created resource, here location means which end point we should hit to get this newly created resource. Here we are adding a location header which contains.
* 
* To Give name we use [HttpGet , Route(“”), Name = “”);
* This will not work if your service is behind load balance as Url.Link generates a route from a server perspective.
* If we add question mark after a route param then it becomes optional param, and then we can have a default value for it. Or it can be defined in the route as well.

Example: [HttpGet, Route(“{id?}”] or [HttpGet, Route(“{id=5}] but with the later approach if we have to pass null we do [HttpGet, Route(“{id=}”]

## Route Order Precedence

1. By Route with order property
2. Segment by segment check by
   1. Literal segments
   2. Parameters with constraints
   3. Parameter without constraints
   4. Wildcard parameters with constraints
   5. Wildcard parameters without constraints.
3. String compare route template

* If we have tried all ways but still cannot resolve the ambiguity then we should use the route order property, the default value of this is 0 for every route. The lower the number the higher the precedence. [HttpGet, Order = 2]

## Note

* Let’s say when we have created a resource and in the response location header we want to add the location of the newly created resource then we can do like following
* Add a name to route for get by Id route -> [Route(“{id:int:min(1), Name=”GetStudentById”)]
* And then in post add this
  + Response.Headers.Location = new Uri(Url.Link(“PassROutenameHere”, new {id = student.id});

## Parameter Binding

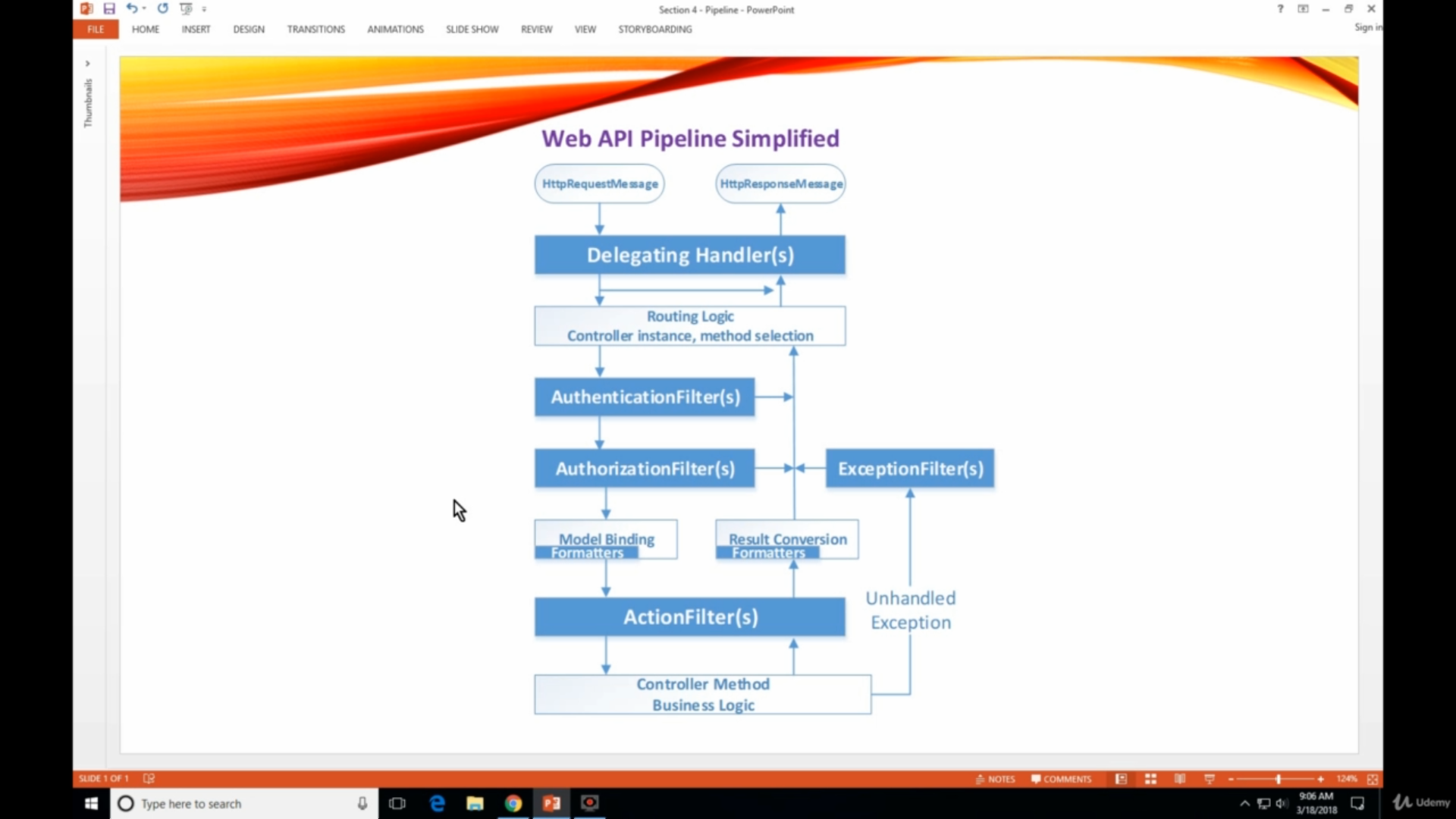
* It is the process of extracting info from http request and converting it to the C# objects.
* We can create a custom binding. We have to implement IModelBinder and implement the method BindModel.
* In model binder we get HttpActionContext as the controller and the action method has already been decided.
* Simple types should be ideal from Uri and the complex type from the body.

## Wildcard Routing

* We can have only a single parameter with a wildcard set and this has to be the last parameter.

# The Web API Pipeline – Delegating Handlers

* .Net Web API has a request and response processsesing pipeline.
* **Very Very Important Request processsesing pipeline -** [**https://www.asp.net/media/4071077/aspnet-web-api-poster.pdf**](https://www.asp.net/media/4071077/aspnet-web-api-poster.pdf)



* We should use delegating handler for something like checking for message level validation such as absence or presence of headers or modify the http request itself. Delegating handlers can quickly abort the pipeline if something is missing in the request hence saving us some performance.
* There is sample source code added with the project itself.

## Delegate Handlers

* All are designed to be async for best performance
* All have a daisy chain pattern to them.
* A message handler is a class that receives an HTTP request and returns an HTTP response. Message handlers derive from the abstract **HttpMessageHandler** class.
* Typically, a series of message handlers are chained together. The first handler receives an HTTP request, does some processing, and gives the request to the next handler. At some point, the response is created and goes back up the chain. This pattern is called a delegating handler.

## Action Filters

* Action Filters are general purpose processor but they execute after the authentication, authorization and model binding has occurred from the inbound side. They have deeper level access of the request like param, controller and action method. They can manipulate the response.
* Action filter can be done per route.
* The ASP.NET MVC framework supports four different types of filters:

1. Authorization filters – Implements the IAuthorizationFilter attribute.
2. Action filters – Implements the IActionFilter attribute.
3. Result filters – Implements the IResultFilter attribute.
4. Exception filters – Implements the IExceptionFilter attribute.

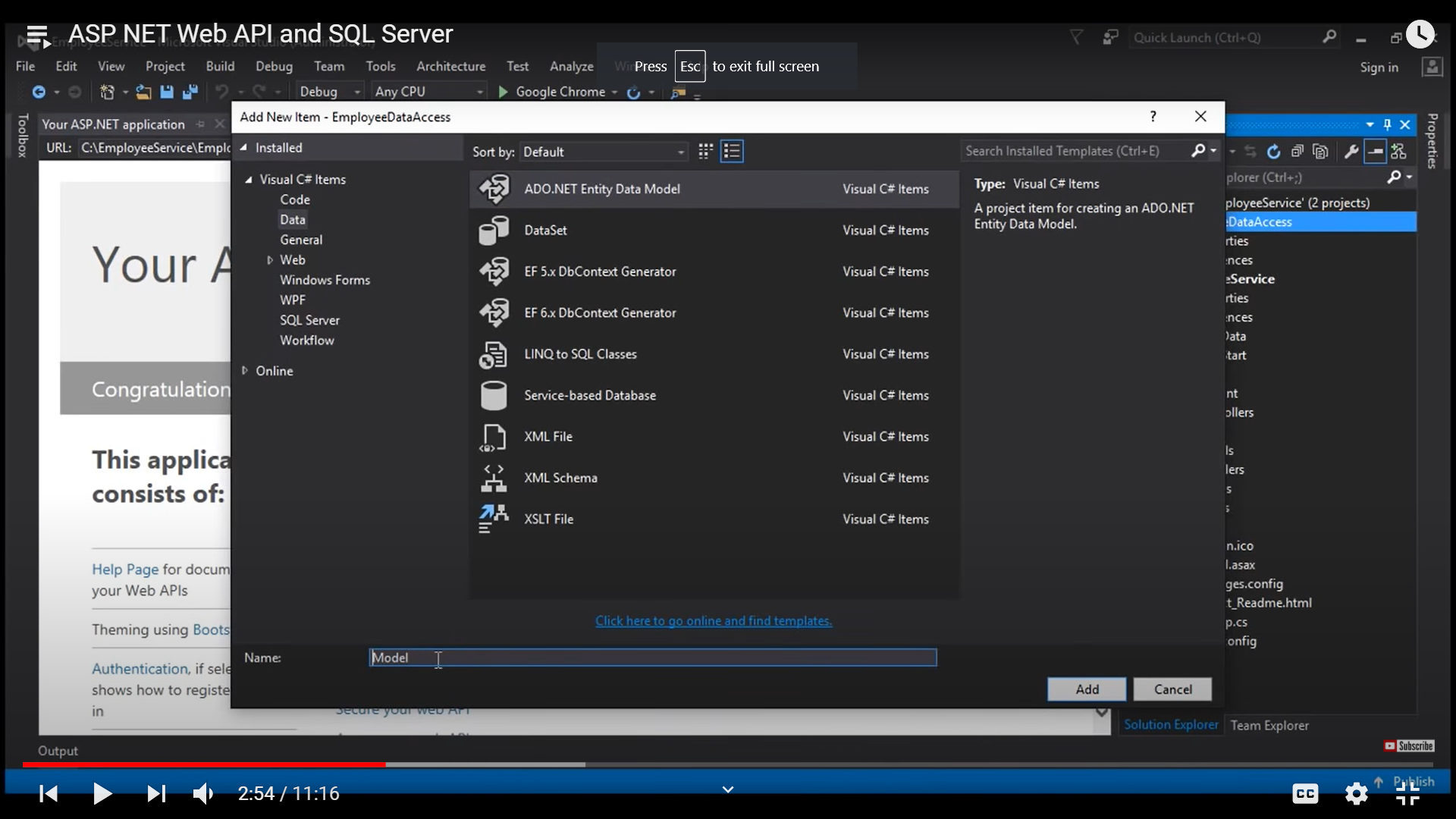
* Filters are executed in the order listed above.
* Action filters contain logic that is executed before and after a controller action executes. You can use an action filter, for instance, to modify the view data that a controller action returns.
* Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.
* Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You also can use exception filters to log errors.
* Each different type of filter is executed in a particular order. If you want to control the order in which filters of the same type are executed then you can set a filter's Order property.
* The base class for all action filters is the System.Web.Mvc.FilterAttribute class. If you want to implement a particular type of filter, then you need to create a class that inherits from the base Filter class and implements one or more of the IAuthorizationFilter, IActionFilter, IResultFilter, or IExceptionFilter interfaces.

## Exception and Error Handler

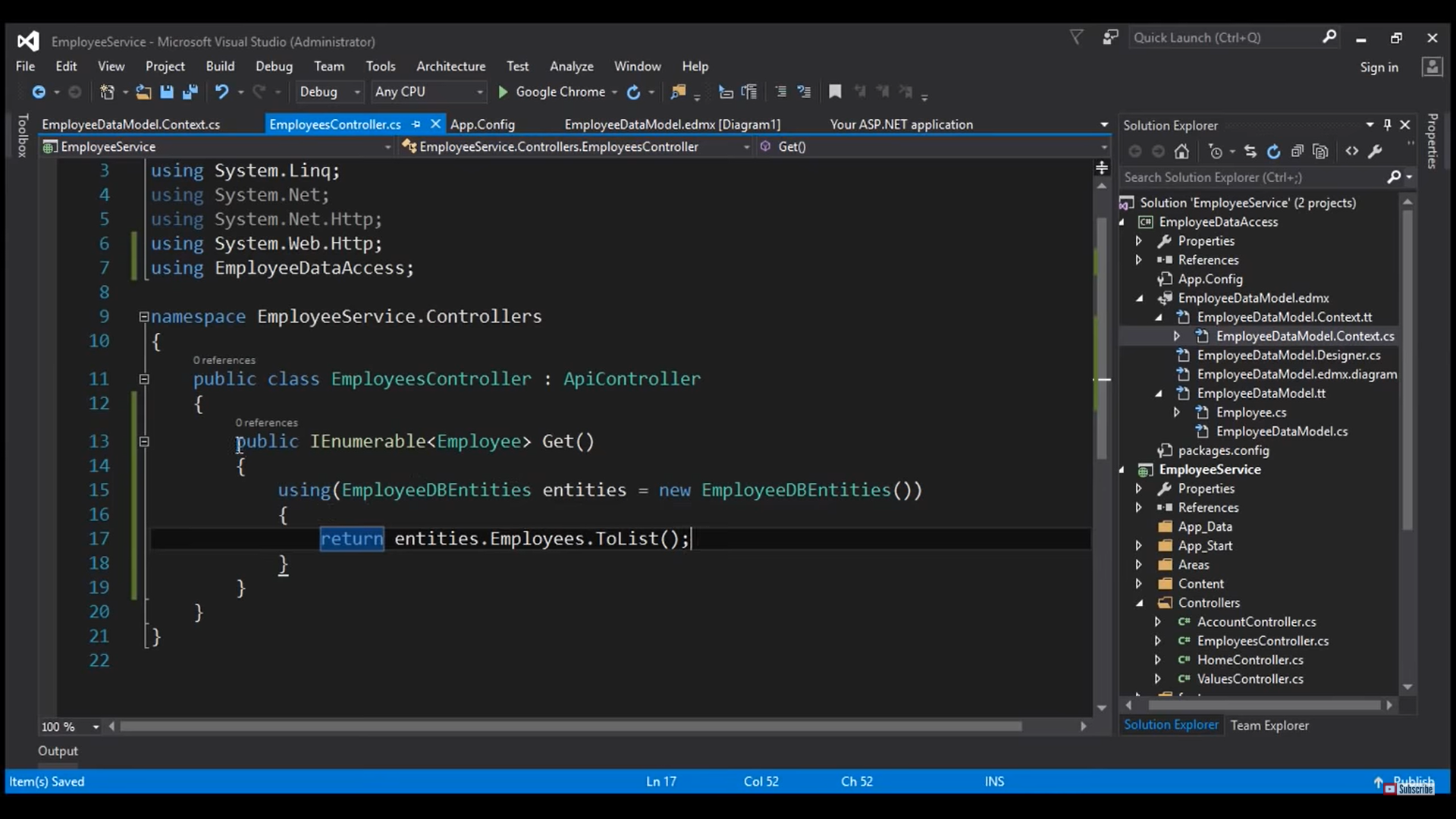
* **TO BE DONE**

## ASP NET WEB API AND SQL SERVER – Data base first approach

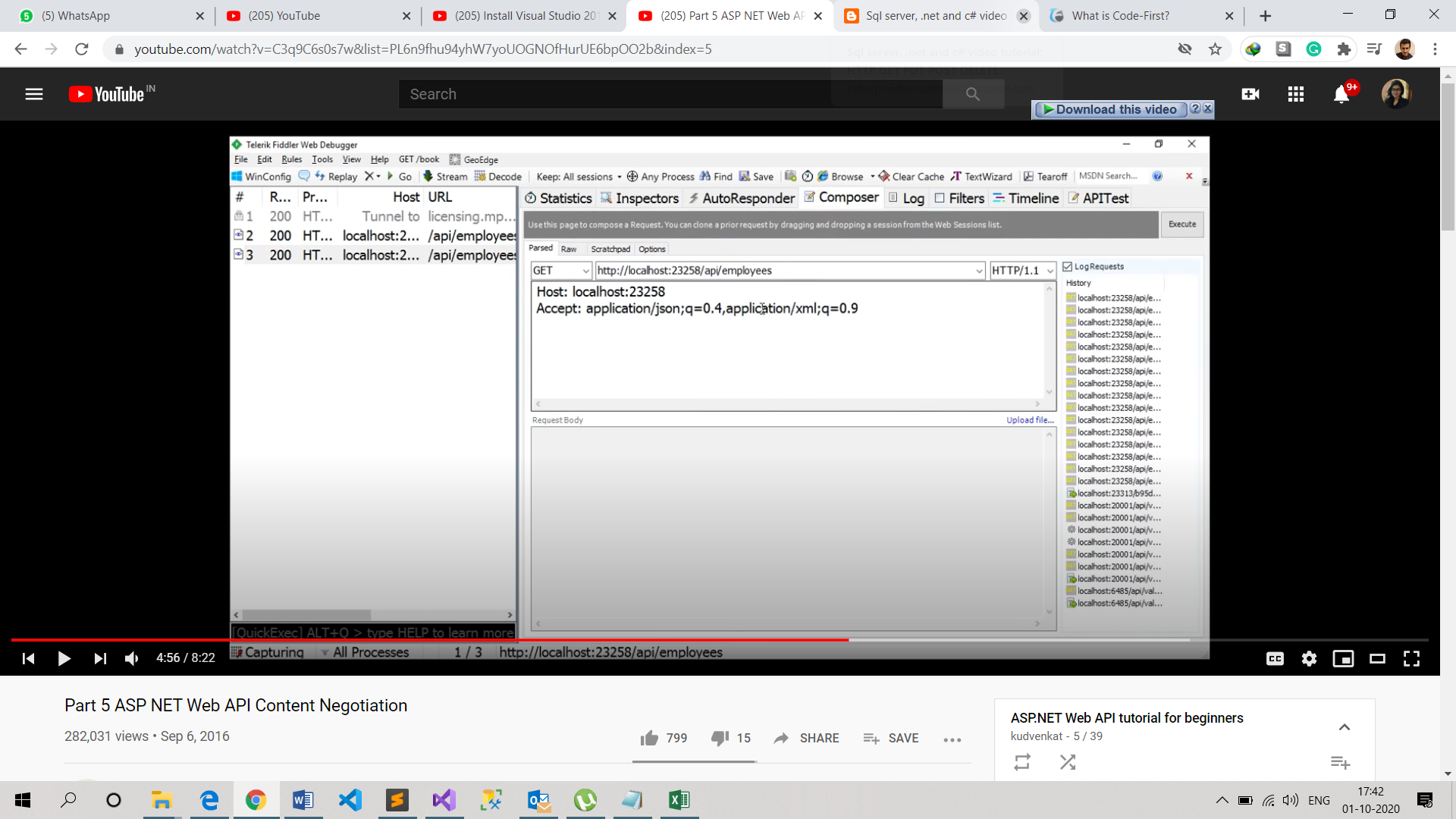
* We will be retrieving data from the SQL Server data base using Web API and Entity Framework.
* For this create your web API Project.
* To your WEB API project and right click and add a ADO.NET Entity Data model.



* The select EF design from the database and select create new connection. And Post this your new project will be created and now Add the reference of this project to your Main WebAPI project.
* Then add the WEB API empty Controller and Call it EmployeesController and add the below method to return the list of all employees. The EmployeesDBENtities is a class generated by default as result of scaffolding. You may also need to add your connection string Werb.config of your web api project.



## Content Negotiation

* One of the property of the web api is that the client should be able to decide the format of the response. The client while sending a request also sends a header called Accept: using which we can mention the format of response from the server.
* The response also has a header where the content-type header is sent to tell what is the response type of the response.
* There are built in media type formatter and we can plugin our own formatter, we can give multiple value for accept header then the server will return in the first matching.
* You can also pass a quality value with the accept so the one with higher will get selected.
* 
* If the Accept header is missing then the default return type of web API will be JSON.
* We can change the serialization setting to change setting like we donot want pascal case we can change it to the camel case.
* Goto WebApiConfig Add following code - >

//This would indent the json response.

config.Formatters.JsonFormatter.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;

config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

## MediaType Formatter

* It is an abstract class from which all the formatter derive from. Eg JsonMediaTypeFormatterClass
* To return only json from WebAPI add the following code to the Register.

// This would ensure the service only returns json and not XML format

config.Formatters.Remove(config.Formatters.XmlFormatter);

* When a request is issued from browser we need json and when it is issued from the any tools like fiddler then we want to return in the requested format so for this we can do the following:

config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));

OR WE HAVE APPROACH 2

Make a class

public class CustomJSONFormatter: JsonMediaTypeFormatter

{

public CustomJSONFormatter()

{

this.SupportedMediaTypes.Add(new System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));

}

public override void SetDefaultContentHeaders(Type type, HttpContentHeaders headers, MediaTypeHeaderValue mediaType)

{

base.SetDefaultContentHeaders(type, headers, mediaType);

headers.ContentType = new MediaTypeHeaderValue("application/json");

}

}

And in register method add following code

config.Formatters.Add(new CustomJSONFormatter());

## Implement POST Method

[HttpPost, Route("")]

public void Post([FromBody] string value)

{

try

{

var message = Request.CreateResponse(HttpStatusCode.Created, value);

message.Headers.Location = new Uri(Request.RequestUri + value);

}

catch

{

var errorMessage = Request.CreateErrorResponse(HttpStatusCode.InternalServerError, "Error");

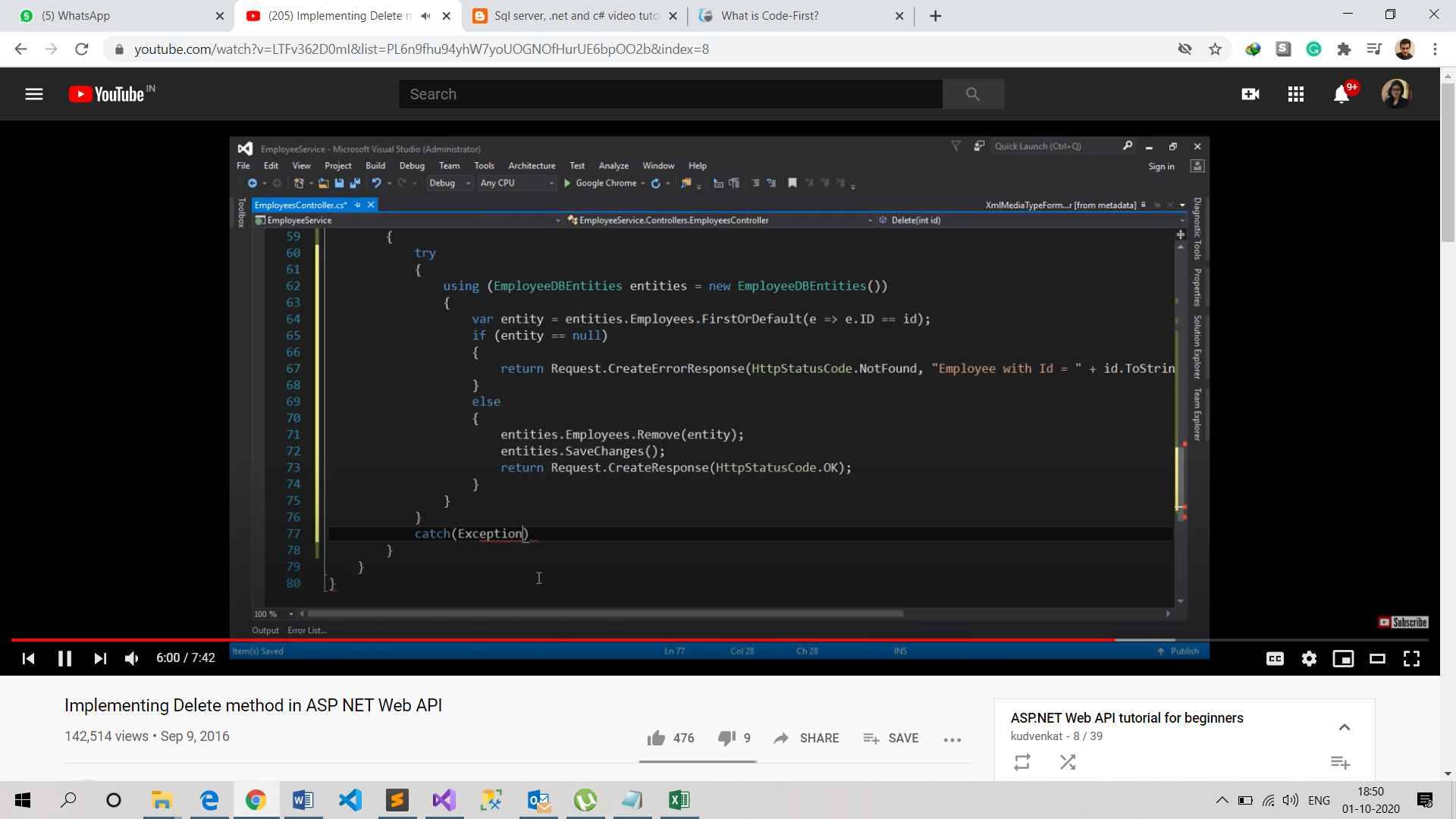
}

}

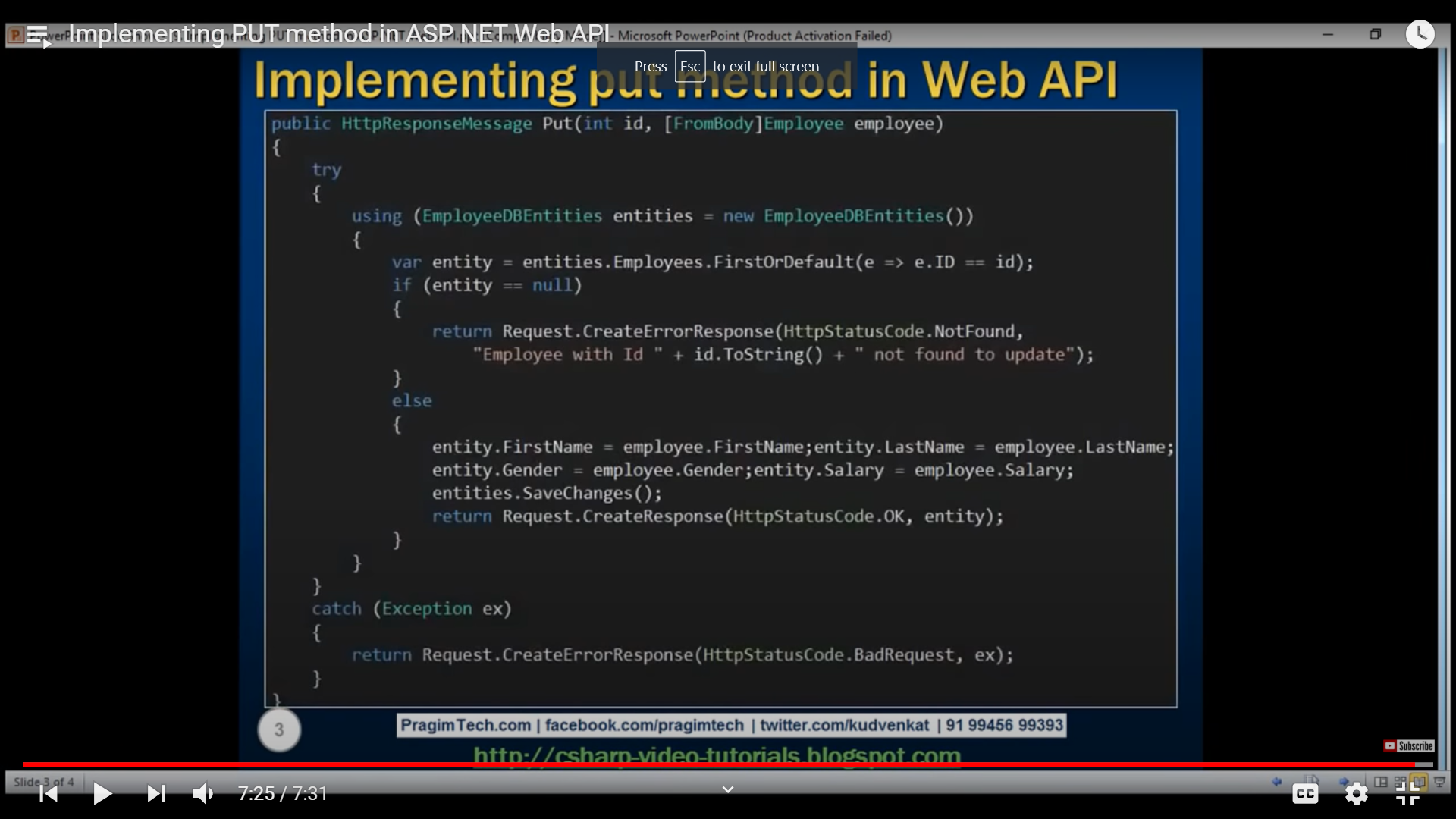
**Important POST Rules**

* If a method return type is void then the default return code should be 204 No Content
* When a new item is created, we should be returning the status code of 201 Created
* With 201 Status code we should return the location of the created resource , i.e URI of the newly created resource.
* When an item is not found instead of returning null and 200 we should return 404 Not Found.

## Implementing Delete Method



## And we can return BadRequest in Catch block Implementing PUT Method



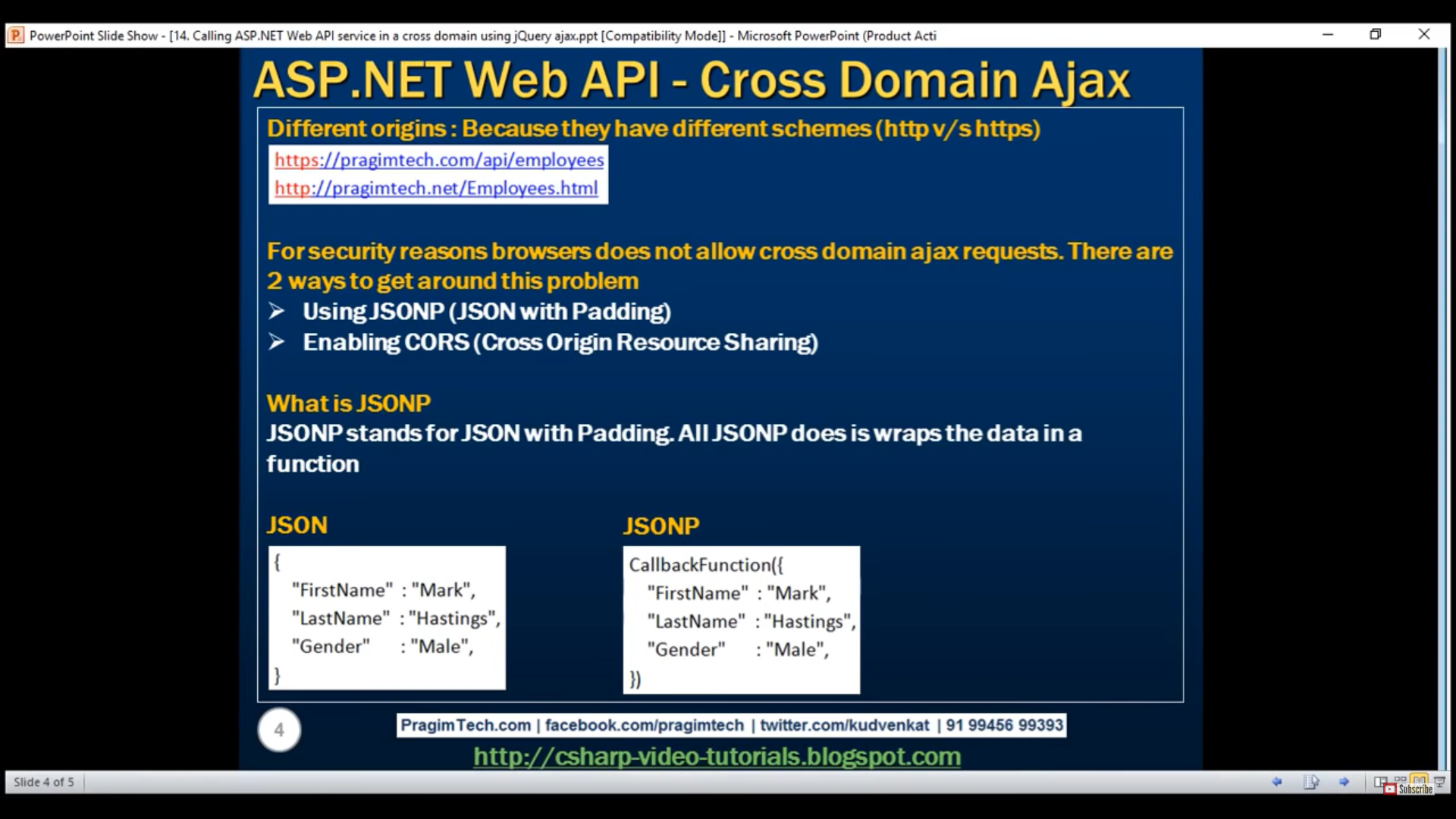
## Query String Parameters in ASP NET Web Api

* If you give a default value to a query param then it makes it optional

## From Body and From Uri Attribute

* By default if the param is of simple type then API will try to get value from the URI either from URI or Query String

## Same Origin Policy

* Browsers allow a web page to request only within the same domain. Browser security prevents a web page from making cross origin requests.
* Origin = Same server + Same Port + same domain + same security (http https)
* There are two ways to solve this problem
  + Using JSONP – Json with Padding
    - All JSONP does is wraps JSON in a JS FUNCTION
    - 
    - TO do this you will need this Nuget Package – WebApiCOntrib.Formatting.JsonP
    - And add code to Register Method.
  + Enabling CORS – Cross Origin Resource Sharing
    - Package – Microsoft.AspNet.WebApi.Cors
    - In Register Method add the following code
    - EnableCorsAttribute cors = new EnableCorsAttribute("\*", "\*", "GET, POST");
    - config.EnableCors(cors);
    - This Enable method accepts – origins, headers and methods.
    - TO enable cors on a single controller using EnableCors attribute at a controller or action method.
    - You can use [DisableCors] attribute on a single action method to disable cors.
    - When cors is enabled, there is origin header which contains site address which issued the header. And there is a Acces-Control-Allow-Origin we have which site the access is granted for.

## Enabling SSL

* To enable SSL using Visual Studio, then goto project property windows and enable the ssl Enabled and set it to use. Then you will get 2 url a sssl url and and a non ssl url
* <https://www.youtube.com/watch?v=4hb6iD3nP6g&list=PL6n9fhu94yhW7yoUOGNOfHurUE6bpOO2b&index=16>
* Lecture 16 to 29 left