Dhruvil Dobariya

dhruvildobariya21@gmail.com

Abstract

In this module I learned advanced concept of API development in ASP.NET

Advanced API

Documentation



Index

[1 Action Result 1](#_Toc133853396)

[1.1 Introduction 1](#_Toc133853397)

[1.2 Void: 1](#_Toc133853398)

[1.3 Entity: 1](#_Toc133853399)

[1.4 HttpResponseMessage 1](#_Toc133853400)

[1.5 IHttpActionResult 2](#_Toc133853401)

[2 CORS 3](#_Toc133853402)

[2.1 Introduction: 3](#_Toc133853403)

[2.2 Enable CORS 3](#_Toc133853404)

[3 Filters 7](#_Toc133853405)

[3.1 Introduction: 7](#_Toc133853406)

[4 HTTP Caching 8](#_Toc133853407)

[4.1 Introduction 8](#_Toc133853408)

[5 Versioning 10](#_Toc133853409)

[5.1 Introduction 10](#_Toc133853410)

[5.2 URL Versioning 10](#_Toc133853411)

[5.3 Query Parameter Versioning 10](#_Toc133853412)

[5.4 Header Versioning: 11](#_Toc133853413)

[5.5 Media Type 11](#_Toc133853414)

1. Action Result

# Introduction

* We have four types of result which is possible to return in ASP.NET web API.
  + HttpResponseMessage
  + IHttpActionResult
  + Void
  + Any type of entity

# Void:

* Void means our action return nothing.

**Example:**

[HttpGet]

*public* *void* Ok()

{

*// code...*

}

# Entity:

* In this case our action return some entity in type of object which should be like, int, string, Product.

**Example:**

[HttpGet]

*public* *List*<*Product*> Ok()

{

*List*<*Product*> products = *new* *List*<*Product*>();

*return* products;

}

# HttpResponseMessage

* HttpResponseMessage represents HTTP Response Message as per MSDN definition.
* If the return type of the action method is one of the Web API's action results, then the API converts the return value to a HTTP Response Message.
* This action result gives us more flexibility to create our own custom message using it's properties.

**Example:**

[HttpGet]

*public* *HttpResponseMessage* Ok()

{

*return* *new* *HttpResponseMessage*()

    {

        Content = *new* *StringContent*("This is content"),

        StatusCode = HttpStatusCode.OK, *// 200*

        RequestMessage = *new* *HttpRequestMessage*(HttpMethod.Get, "request uri")

    };

}

# IHttpActionResult

* The IHttpActionResult action result was introduced in Web API 2.
* It also returns the HttpResponseMessage. But the code we write to send the HTTP Response Message will be reduced with this interface.
* The preceding code returns the HTTP Response message as 200 Ok Status Code.
* As per IHttpActionResult's definition in the ASP.Net/Web API, it acts like a factory for HttpResponseMessage and comes with built-in responses, like Ok, BadRequest, NotFound, Unauthorized, Exception, Conflict and Redirect.

**Example:**

[HttpGet]

*public* *IHttpActionResult* Ok()

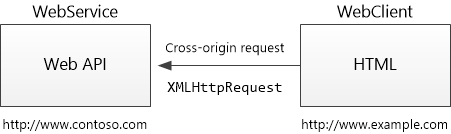
{

*return* Ok("This is Content");

}

1. CORS

# Introduction:

* When we use web API and frontend on different origins at that time CORS come in picture.
* CORS means cross origin resource sharing.
* For security reason browser restrict the cross origin requests.
* CORS allows us to access cross origin resource.
* Here is the example of same origin,
  + http://example.com/foo.html
  + http://example.com/bar.html
* Here is the example of cross origin,
  + http://example.net - Different domain
  + http://example.com:9000/foo.html - Different port
  + https://example.com/foo.html - Different scheme
  + http://www.example.com/foo.html - Different subdomain

# Enable CORS

* We can enable CORS using “Microsoft.AspNet.WebApi.Cors” package.
* We can configure CORS using EnableCorsAttribute at action level, controller level and global level.
* We have four properties in for CORS.

**Origins:**

* Here, we need to set Origins which means from which domain the requests will accept.
* If we have more than one domain, then you can set as comma separated.
* Additionally, if you want any domain request to be accepted then use wild card as "\*".

**Request Headers:**

* The Request header parameter specifies which Request headers are allowed.
* To allow any header set value to "\*".

**HTTP Methods:**

* The methods property specifies which HTTP methods are allowed to access the resource.
* We can use comma-separated values when you have multiple HTTP methods like "get, put, post".
* When we want to allow all HTTP methods, then we should use the wildcard value "\*".

**Exposed Headers:**

* By default, the browser does not expose all of the response headers to the application.
* Which mean browser only give access to client these headers which is exist in default set like, Cache-Control, Content-Language, Content-Type, Expires, Last-Modified, Pragma.
* Which means if we add any custom header in response then it don’t accessed by client.
* So exposed header should help us to accessed our custom header by client which requested from cross origin.

## Action Level:

* Here we use EnableCors attribute on action method.
* Action level configuration override configuration of controller level as well as global level configuration.

**Example:**

*public* *HttpResponseMessage* Get(*int* *id*, *string* *name*)

{

    HttpContext.Current.Response.Headers.Add("X-Id", *id*.ToString());

    HttpContext.Current.Response.Headers.Add("X-Name", *name*);

*return* *new* *HttpResponseMessage*()

    {

        Content = *new* *StringContent*(JsonConvert.SerializeObject(*new* {*id*,*name*})),

        StatusCode = HttpStatusCode.OK

    };

}

## Controller Level:

* Here we use EnableCors attribute on controller, so it applied all the method of controller.
* Controller level configuration override global level configuration.

**Example:**

[EnableCors(origins: "www.something1.com, www.somthing2.com", headers: "X-Header1, X-Header2", methods: "GET,PUT,POST,DELETE", exposedHeaders: "X-Id,X-Name")]

*public* *class* HomeController : ApiController

{

    [HttpGet]

*public* *HttpResponseMessage* Get(*int* *id*, *string* *name*)

    {

        HttpContext.Current.Response.Headers.Add("X-Id", *id*.ToString());

        HttpContext.Current.Response.Headers.Add("X-Name", *name*);

*return* *new* *HttpResponseMessage*()

        {

            Content = *new* *StringContent*(JsonConvert.SerializeObject(*new* { *id*, *name* })),

            StatusCode = HttpStatusCode.OK

        };

    }

    [HttpPost]

*public* *HttpResponseMessage* Set(*int* *id*, *string* *name*)

    {

        HttpContext.Current.Response.Headers.Add("X-Id", *id*.ToString());

        HttpContext.Current.Response.Headers.Add("X-Name", *name*);

*return* *new* *HttpResponseMessage*()

        {

            Content = *new* *StringContent*(JsonConvert.SerializeObject(*new* { *id*, *name* })),

            StatusCode = HttpStatusCode.OK

        };

    }

}

## Global Level:

* Here we configure COES in WebConfig.cs file.
* This configuration applied on whole application.

**Example:**

*using* System.Web.Http;

*using* System.Web.Http.Cors;

*namespace* CORSLearn

{

*public* *static* *class* WebApiConfig

    {

*public* *static* *void* Register(*HttpConfiguration* config)

        {

*// Web API configuration and services*

*// CORS configuration*

*EnableCorsAttribute* cors = *new* *EnableCorsAttribute*("\*", "\*", "\*", "X-Id, X-Name");

            config.EnableCors(cors);

*// Web API routes*

            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(

                name: "DefaultApi",

                routeTemplate: "api/{controller}/{action}/{id}",

                defaults: *new* { id = RouteParameter.Optional }

            );

        }

    }

}

1. Filters

# Introduction:

* Web API includes filters to add extra logic before or after action method executes.
* Filters can be used to provide cross-cutting features such as logging, exception handling, performance measurement, authentication and authorization.
* Filters are actually attributes that can be applied on the Web API controller or one or more action methods.
* Every filter attribute class must implement IFilter interface included in System.Web.Http.Filters namespace.
* However, System.Web.Http.Filters includes other interfaces and classes that can be used to create filter for specific purpose.
* We have different types of filter,

|  |  |  |  |
| --- | --- | --- | --- |
| Filter Type | Interface | Class | Description |
| Simple Filter | IFilter | - | Defines the methods that are used in a filter |
| Action Filter | IActionFilter | ActionFilterAttribute | Used to add extra logic before or after action methods execute. |
| Authentication Filter | IAuthenticationFilter | - | Used to force users or clients to be authenticated before action methods execute. |
| Authorization Filter | IAuthorizationFilter | AuthorizationFilterAttribute | Used to restrict access to action methods to specific users or groups. |
| Exception Filter | IExceptionFilter | ExceptionFilterAttribute | Used to handle all unhandled exception in Web API. |
| Override Filter | IOverrideFilter | - | Used to customize the behaviour of other filter for individual action method. |

* All filters have two types of methods,
  + OnExecuting – which called before task
  + OnExecuted – which called after task

1. HTTP Caching

# Introduction

* In ASP.NET Web API, HTTP caching is a technique used to improve performance and reduce network traffic by storing and reusing previously retrieved HTTP responses.
* It allows clients (such as web browsers or other API consumers) to cache the responses of API requests and use them for subsequent requests instead of making a new request to the server.
* HTTP caching is based on the cache-related headers defined in the HTTP protocol.
* Here are some commonly used cache-related headers:

**Cache-Control:**

* This header specifies caching directives for both the client and intermediary caching servers.
* It can include directives like "public" (indicating that the response can be cached by any client), "private" (indicating that the response is specific to a particular client), "max-age" (indicating the maximum time the response can be cached), and more.

**Expires:**

* This header specifies an expiration date and time after which the response should be considered stale and no longer valid.
* It is an alternative to the "Cache-Control" header.

**ETag:**

* This header provides a unique identifier for a specific version of a resource.
* It allows clients to send the ETag value in subsequent requests using the "If-None-Match" header.
* If the resource hasn't changed (as indicated by the ETag), the server can respond with a "304 Not Modified" status code, indicating that the cached response can be used.

**Last-Modified:**

* This header indicates the last modified date and time of the resource.
* It is used in conjunction with the "If-Modified-Since" header sent by the client in subsequent requests.
* If the resource hasn't been modified since the provided date, the server can respond with a "304 Not Modified" status code.
* To enable HTTP caching in ASP.NET Web API, you can use the CacheCow library, which provides an implementation of the HTTP caching standards.
* It allows you to apply caching attributes to API methods or controllers to control the caching behaviour.
* Additionally, you can manually set the cache-related headers in your code using the HttpResponseMessage object.
* It's important to note that caching is effective for static or relatively static resources.
* For dynamic content, we may need to implement additional strategies such as conditional caching or cache invalidation to ensure that clients receive up-to-date data when necessary.

1. Versioning

# Introduction

* Versioning in ASP.NET Web API 2 refers to the practice of managing and supporting multiple versions of an API.
* It allows you to introduce changes to your API while ensuring backward compatibility for existing clients.
* This way, you can evolve your API over time without breaking existing client applications.
* We have four commonly used approaches,
* You can also consider using external libraries or frameworks to help with versioning, such as Microsoft's Microsoft.AspNet.WebApi.Versioning package, which provides additional features and options for versioning your Web API.
  + Url versioning
  + Query parameter versioning
  + Header versioning:
  + Media type

# URL Versioning

* In this approach, the version number is included in the URL of the API endpoint.
* For example:
  + https://api.example.com/v1/products
  + https://api.example.com/v2/products
* This method is straightforward and easy to understand, but it can lead to longer URLs and can be cumbersome if you have many versions or frequent changes.

# Query Parameter Versioning

* With query parameter versioning, the version number is specified as a query parameter in the API URL.
* For example:
  + https://api.example.com/products?version=1
  + https://api.example.com/products?version=2
* This approach keeps the base URL clean and allows for simpler routing.
* However, it may require additional effort to parse and handle the version parameter in your code.

# Header Versioning:

* In header versioning, the version number is sent as a custom header in the HTTP request.
* For example:
  + GET /products HTTP/1.1
  + Host: api.example.com
  + Accept: application/json
  + Api-Version: 1
* This method keeps the URL clean and doesn't affect the routing.
* It allows for more flexibility and doesn't require modifying the URL structure.
* However, it may require additional effort to handle and interpret the custom header in your code.

# Media Type

* Certainly! Media type versioning, also known as content negotiation or MIME type versioning, is another approach to versioning in ASP.NET Web API 2.
* In this approach, the version number is embedded in the media type (MIME type) of the request or response.
* There are two commonly used methods for media type versioning,
  + Media Type Parameter Versioning
  + Media Type Subtype Versioning

## Media Type Parameter Versioning:

* In this approach, a version parameter is added to the media type as a parameter.
* For example:
  + Accept: application/json; version=1.0
  + Content-Type: application/json; version=2.0
* By including the version as a parameter in the media type, the client can specify the desired version in the request, and the server can respond accordingly.
* This method allows for explicit versioning and precise control over the version specified.

## Media Type Subtype Versioning:

* In this approach, the version number is embedded as a subtype within the media type.
* For example:
  + Accept: application/versionlearn.v1+json
  + Content-Type: application/versionlearn.v2+json
* The version number is included as a separate subtype, denoted by the "v1" or "v2" in the examples above.
* This method provides a clear distinction between different versions and can be useful when we have significant changes between versions.