EBook Gratuito

APPRENDIMENTO

asp.net-web-api

Free unaffiliated eBook created from
Stack Overflow contributors.

#asp.net-web-api

Sommario

Di

Capitolo 1: Iniziare con asp.net-web-api

Osservazioni

Questa sezione fornisce una panoramica di ciò che asp.net-web-api è, e perché uno sviluppatore potrebbe voler usarlo.

Dovrebbe anche menzionare qualsiasi argomento di grandi dimensioni all'interno di asp.net-web-api e collegarsi agli argomenti correlati. Poiché la documentazione di asp.net-web-api è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Examples

Installazione o configurazione

Istruzioni dettagliate su come installare o installare asp.net-web-api.

Cosa e perché API Web ASP.NET?

Che cosa? : Un framework completamente supportato ed estensibile per la creazione di endpoint basati su HTTP. Nel mondo di HTML5, dispositivi mobili e moderne tecniche di sviluppo, HTTP è diventato l'opzione predefinita per la creazione di servizi ricchi e scalabili. L'API Web ASP.NET fornisce un set di opzioni predefinite di facile utilizzo, ma fornisce anche un'infrastruttura di estensibilità profonda per soddisfare le esigenze di qualsiasi scenario che utilizza HTTP.

Perché? :

- Un'applicazione HTML5 che richiede un livello di servizi.
- Un'applicazione mobile che ha bisogno di un livello di servizi.
- Un'applicazione desktop client-server che richiede un livello di servizi.

Per aggiungere Web API a un'applicazione MVC esistente.

Utilizzare Nuget per trovare il pacchetto Web Api.

Puoi farlo usando i pacchetti Gestisci Nuget e cercando il pacchetto Web Api o usa Nuget Package Manager e digita

```
PM> Install-Package Microsoft.AspNet.WebApi
```

Aggiungi WebApiConfig.cs alla cartella App_Start / Il file di configurazione dovrebbe contenere questo.

```
using System.Web.Http;
namespace WebApplication1
{
```

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(config =>
        {
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        });
    }
 }
}
```

Origine: configurazione dell'API Web ASP.NET

Aggiungi `GlobalConfiguration.Configure(WebApiConfig.Register);` in Application_Start del file Global.asax.

Leggi Iniziare con asp.net-web-api online: https://riptutorial.com/it/asp-net-web-api/topic/1058/iniziare-con-asp-net-web-api

Capitolo 2: Abilitazione di CORS API WEB ASP.NET

Examples

Abilitazione di CORS per WebAPI 2

```
// Global.asax.cs calls this method at application start
public static void Register(HttpConfiguration config)
{
    // New code
    config.EnableCors();
}

//Enabling CORS for controller after the above registration
[EnableCors(origins: "http://example.com", headers: "*", methods: "*")]
public class TestController : ApiController
{
    // Controller methods not shown...
}
```

Abilitazione CORS a livello globale per Web API 2

```
public static void Register(HttpConfiguration config)
{
    var corsAttr = new EnableCorsAttribute("http://example.com", "*", "*");
    config.EnableCors(corsAttr);
}
```

Abilitazione di CORS in Asp.Net 5 per tutti i domini e metodi

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(o => o.AddPolicy("MyPolicy", builder =>
    {
        builder.AllowAnyOrigin()
               .AllowAnyMethod()
               .AllowAnyHeader();
    }));

    // ...
}

public void Configure(IApplicationBuilder app)
{
    app.UseCors("MyPolicy");

    // ...
}
```

Abilitazione di CORS in Asp.Net 5 per domini e metodi specifici

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddCors();
    services.ConfigureCors(options =>
         options.AddPolicy("AllowSpecific", p => p.WithOrigins("http://localhost:1233")
                                                  .WithMethods("GET")
                                                  .WithHeaders("name")));
}
```

Configura CORS per WebAPI 2 con l'autenticazione di Windows

La seguente configurazione lato server consente alla richiesta CORS di funzionare insieme all'autenticazione di Windows (non è necessario abilitare l'anonimato in IIS).

web.config - consente le richieste di verifica preliminare non autenticate (anonime) (OPZIONI)

```
<system.web>
    <authentication mode="Windows" />
    <authorization>
        <allow verbs="OPTIONS" users="*"/>
        <deny users="?" />
    </authorization>
</system.web>
```

global.asax.cs - rispondi correttamente con le intestazioni che consentono al chiamante di un altro dominio di ricevere dati

```
protected void Application_AuthenticateRequest(object sender, EventArgs e)
{
    if (Context.Request.HttpMethod == "OPTIONS")
    {
        if (Context.Request.Headers["Origin"] != null)
            Context.Response.AddHeader("Access-Control-Allow-Origin",
Context.Request.Headers["Origin"]);

        Context.Response.AddHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept, MaxDataServiceVersion");
        Context.Response.AddHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE,
OPTIONS");
        Context.Response.AddHeader("Access-Control-Allow-Credentials", "true");

        Response.End();
    }
}
```

Abilitazione CORS

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
```

```
        // all requests are enabled in this example. SupportsCredentials must be here to allow
authenticated requests
        var corsAttr = new EnableCorsAttribute("*", "*", "*") { SupportsCredentials = true };
        config.EnableCors(corsAttr);
    }
}

protected void Application_Start()
{
    GlobalConfiguration.Configure(WebApiConfig.Register);
}
```

Invia correttamente la richiesta autenticata da jQuery con l'endpoint Web API 2

L'esempio seguente mostra come costruire correttamente entrambe le richieste GET e POST contro Web API 2 (CORS deve essere configurato lato server, se inviato da un altro dominio):

```
<script type="text/javascript" src="https://code.jquery.com/jquery-3.1.1.js"></script>
CORS with Windows Authentication test
<script type="text/javascript">

    // GET
    $.ajax({
        url: "endpoint url here",
        type: "GET",
        dataType: "json",
            xhrFields: {
            withCredentials: true
        }
    })
    .done(function (data, extra) {
      alert("GET result" + JSON.stringify(data));
    })
    .fail(function(data, extra) {
    });

    //POST
    $.ajax({
        url: "url here",
        type: "POST",
        contentType: 'application/json; charset=utf-8',
        data: JSON.stringify({testProp: "test value"}),
        xhrFields: {
            withCredentials: true
        },
        success: function(data) {
            alert("POST success - " + JSON.stringify(data));
        }
    })
    .fail(function(data) {
        alert("Post error: " + JSON.stringify(data.data));
    });

</script>
```

Codice lato server:

```
    [System.Web.Http.HttpGet]
    [System.Web.Http.Route("GetRequestUsername")]
    public HttpResponseMessage GetRequestUsername()
    {
        var ret = Request.CreateResponse(
            HttpStatusCode.OK,
            new { Username = SecurityService.GetUsername() });
        return ret;
    }

    [System.Web.Http.HttpPost]
    [System.Web.Http.Route("TestPost")]
    public HttpResponseMessage TestPost([FromBody] object jsonData)
    {
        var ret = Request.CreateResponse(
            HttpStatusCode.OK,
            new { Username = SecurityService.GetUsername() });
        return ret;
    }
```

Invia correttamente la richiesta autenticata da AngularJS contro l'endpoint Web API 2

```
<script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.1/angular.js"></script>
CORS with Windows Authentication test (Angular)
<script type="text/javascript">

    var app = angular.module('myApp', []);
    app.controller('myCtrl', function($http) {

        $http(
            {
                method: 'GET',
                url: 'url here',
                withCredentials: true,
            }
        )
        .then(function(data) {
            alert("Get result = " + JSON.stringify(data.data));
        },
        function(data, extra) {
            alert("Get failed: " + JSON.stringify(data.data));
        });

        $http(
            {
                method: 'POST',
                url: "url here",
                withCredentials: true,
                data: { url: "some url", message: "some message", type: "some type"}
            }
        )
        .then(function(data) {
            alert("POST success - " + JSON.stringify(data.data));
        },
        function(data) {
            alert("POST failed: " + JSON.stringify(data.data));
        });
```

```
    });

</script>

<div ng-app="myApp" ng-controller="myCtrl">
</div>
```

Leggi Abilitazione di CORS API WEB ASP.NET online: https://riptutorial.com/it/asp-net-web-api/topic/4185/abilitazione-di-cors-api-web-asp-net

Capitolo 3: API Web ASP.NET MediaTypeFormatter

Examples

MediaTypeFormatter Informazioni di base

`MediaTypeFormatter` è una classe astratta da cui ereditano le classi `JsonMediaTypeFormatter` e `XmlMediaTypeFormatter` . Qui, la classe `JsonMediaTypeFormatter` gestisce gli oggetti JSON e la classe `XmlMediaTypeFormatter` gestisce gli oggetti XML.

Restituisce solo JSON indipendentemente dal valore Accept Header:

Per restituire solo oggetti JSON nella risposta della richiesta weather Accept Valore intestazione della richiesta se `application/json` o `application/xml` scrivono la seguente riga nel metodo `Register` della classe `WebApiConfig` .

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

Qui, `config` è un oggetto della classe `HttpConfiguration` . Questa riga di codice rimuove completamente `XmlFormatter` che forza l'API Web ASP.NET a restituire sempre JSON indipendentemente dal valore di intestazione Accept nella richiesta del client. Utilizzare questa tecnica quando si desidera che il servizio supporti solo JSON e non XML.

Restituisce solo XML indipendentemente dal valore Accept Header:

Per restituire solo oggetti XML nella risposta della richiesta meteo Accetta il valore Intestazione della richiesta se `application/json` o `application/xml` scrivono la seguente riga nel metodo `Register` della classe `WebApiConfig` .

```
config.Formatters.Remove(config.Formatters.JsonFormatter);
```

Qui, `config` è un oggetto della classe `HttpConfiguration` come descritto sopra. Questa riga di codice rimuove completamente `JsonFormatter` che forza l'API Web ASP.NET a restituire sempre XML indipendentemente dal valore di intestazione Accept nella richiesta del client. Utilizzare questa tecnica quando si desidera che il proprio servizio supporti solo XML e non JSON.

Restituisci JSON anziché XML:

1. Quando viene emessa una richiesta dal browser, il servizio API Web deve restituire JSON anziché XML.
2. Quando una richiesta viene emessa da uno strumento come il violinista, il valore di intestazione Accept deve essere rispettato. Ciò significa che se l'intestazione Accept è impostata su application / xml il servizio deve restituire XML e se è impostato su application / json il servizio dovrebbe restituire JSON.

Metodo 1:

Includere la seguente riga nel metodo `Register` della classe `WebApiConfig` .

```
config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
MediaTypeHeaderValue("text/html"));
```

In questo modo l'API Web ASP.NET viene utilizzata per utilizzare `JsonFormatter` quando viene richiesta la richiesta di `text/html` che è l'impostazione predefinita per la maggior parte dei browser. Il problema con questo approccio è che l'intestazione `Content-Type` della risposta è impostata su `text/html` che è fuorviante.

Metodo 2:

Usa i formattatori personalizzati. Creare una classe derivata dalla classe `JsonMediaTypeFormatter` e implementare il metodo `SetDefaultContentHeaders` .

Ecco l'esempio della classe di formattazione JSON personalizzata che restituisce il formato JSON in risposta.

```
public class CustomJsonFormatter : JsonMediaTypeFormatter
{
    public CustomJsonFormatter()
    {
        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));
    }

    public override void SetDefaultContentHeaders(Type type, HttpContentHeaders headers,
MediaTypeHeaderValue mediaType)
    {
        base.SetDefaultContentHeaders(type, headers, mediaType);
        headers.ContentType = new MediaTypeHeaderValue("application/json");
    }
}
```

E questo è l'esempio del formattatore del tipo di supporto personalizzato che restituisce il formato CSV in risposta.

```
public class CSVMediaTypeFormatter : MediaTypeFormatter {

    public CSVMediaTypeFormatter()
    {
        SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
    }

    public CSVMediaTypeFormatter(MediaTypeMapping mediaTypeMapping) : this()
    {
        MediaTypeMappings.Add(mediaTypeMapping);
    }

    public CSVMediaTypeFormatter(IEnumerable<MediaTypeMapping> mediaTypeMappings) : this()
    {
        foreach (var mediaTypeMapping in mediaTypeMappings)
        {
            MediaTypeMappings.Add(mediaTypeMapping);
```

```
        }
    }
}
```

Successivamente, implementando la classe di formattazione personalizzata, registrarlo nel metodo `Register` della classe `WebApiConfig` .

```
config.Formatters.Add(new CustomJsonFormatter());
```

Ora, secondo il tuo formattatore, riceverai risposta e `Content-Type` dal server.

Leggi API Web ASP.NET MediaTypeFormatter online: https://riptutorial.com/it/asp-net-web-api/topic/7673/api-web-asp-net-mediatypeformatter

Capitolo 4: Avvio rapido: lavorare con JSON

Osservazioni

Esempi per iniziare e funzionare rapidamente (e correttamente) con Web API ASP.NET

Examples

Restituisci JSON da GET usando gli attributi

1. Configura il tuo formattatore e routing nel `Register` di (`App_Start/WebApiConfig`)

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        GlobalConfiguration.Configuration.Formatters.Clear();
        GlobalConfiguration.Configuration.Formatters.Add(new JsonMediaTypeFormatter());

        config.MapHttpAttributeRoutes();
    }
}
```

2. Creare metodi in un `ApiController`

```
public class HelloWorldController : ApiController
{
  [HttpGet]
  [Route("echo/{message}")]
  public IHttpActionResult Echo(string message) {
    return Ok(new{ hello: message });
  }

  [HttpGet]
  [Route("echo/{digits:int}")]
  public IHttpActionResult Echo(int digits) {
    return Ok(new{ hello: digits });

}
```

eseguendo `GET /echo/foo`

```
{
  "hello": "foo"
}
```

eseguendo `GET /echo/1241290805`

```
{
  "hello": 1241290805
}
```

come il framework di routing prende le condizioni più specifiche (tipo di dati) quando si sceglie un metodo

Leggi Avvio rapido: lavorare con JSON online: https://riptutorial.com/it/asp-net-web-api/topic/3851/avvio-rapido--lavorare-con-json

Capitolo 5: caching

Osservazioni

Il caching è il processo di memorizzazione dei dati da qualche parte per le future richieste, nel nostro caso possiamo evitare il colpo indesiderato al database per ottenere i dati se memorizziamo i dati da qualche parte, in questo modo possiamo assicurarci che i dati siano serviti in maniera più veloce .

Examples

System.Runtime.Caching (MemoryCache)

Importa lo spazio dei nomi System.Runtime.Caching (Assicurati di aver aggiunto System.Runtime.Caching DLL al riferimento del tuo progetto).

Creare un'istanza della classe MemoryCache.

```
MemoryCache memCache = MemoryCache.Default;
```

Aggiungi valori a MemoryCache

```
public IQueryable<tblTag> GettblTags()
        {
            var ca = db.tblTags;
            memCache.Add("tag", ca, DateTimeOffset.UtcNow.AddMinutes(5));
            return db.tblTags;
        }
```

Qui "tag" è la mia chiave e "ca" sono i miei valori e DateTimeOffset.UtcNow.AddMinutes (5) serve per impostare la cache per cinque minuti da ora.

Ottieni valori da MemoryCache

```
var res = memCache.Get("tag");
            if (res != null)
            {
                return res;
            }
            else {
                var ca = db.tblTags;
                memCache.Add("tag", ca, DateTimeOffset.UtcNow.AddMinutes(5));
                return db.tblTags;
            }
```

Otterremo i valori della cache nella variabile res, ricorda che questi valori saranno lì solo per cinque minuti. Puoi sempre modificarlo secondo necessità. Se il valore non è nullo, lo restituiremo e faremo la manipolazione e se è nullo andremo avanti e recupereremo i dati dal database e

aggiungeremo il valore alla cache.

Rimuovi i valori da MemoryCache

```
            if (memCache.Contains("tag"))
            {
                memCache.Remove("tag");
            }
```

Leggi caching online: https://riptutorial.com/it/asp-net-web-api/topic/7983/caching

Capitolo 6: Configura un'applicazione API Web per rispondere con dati JSON belli / formattati di defualt

Examples

Formattazione JSON predefinita: efficienza al costo della leggibilità

Diciamo che hai un semplice ApiController come questo:

```
    [HttpGet]
    [Route("test")]
    public dynamic Test()
    {
        dynamic obj = new ExpandoObject();
        obj.prop1 = "some string";
        obj.prop2 = 11;
        obj.prop3 = "another string";

        return obj;
    }
```

La rappresentazione JSON risultante di questo oggetto sarà simile a questa:

```
{"prop1":"some string","prop2":11,"prop3":"another string"}
```

Questo probabilmente va bene per risposte semplici come questa, ma immagina se hai un oggetto grande / complesso inviato come risposta:

```
"response": { "version": "0.1", "termsofService":
"http://www.wunderground.com/weather/api/d/terms.html", "features": { "history": 1 } },
"history": { "date": { "pretty": "July 16, 2016", "year": "2016", "mon": "07", "mday": "16",
"hour": "12", "min": "00", "tzname": "America/Indianapolis" }, "utcdate": { "pretty": "July
16, 2016", "year": "2016", "mon": "07", "mday": "16", "hour": "16", "min": "00", "tzname":
"UTC" }, "observations": [{ "date": { "pretty": "12:15 AM EDT on July 16, 2016", "year":
"2016", "mon": "07", "mday": "16", "hour": "00", "min": "15", "tzname": "America/Indianapolis"
}, "utcdate": { "pretty": "4:15 AM GMT on July 16, 2016", "year": "2016", "mon": "07", "mday":
"16", "hour": "04", "min": "15", "tzname": "UTC" }, "tempm": "18.2", "tempi": "64.8",
"dewptm": "16.4", "dewpti": "61.5", "hum": "89", "wspdm": "9.3", "wspdi": "5.8", "wgustm": "-
9999.0", "wgusti": "-9999.0", "wdird": "20", "wdire": "NNE", "vism": "16.1", "visi": "10.0",
"pressurem": "1018.2", "pressurei": "30.07", "windchillm": "-999", "windchilli": "-999",
"heatindexm": "-9999", "heatindexi": "-9999", "precipm": "-9999.00", "precipi": "-9999.00",
"conds": "Clear", "icon": "clear", "fog": "0", "rain": "0", "snow": "0", "hail": "0",
"thunder": "0", "tornado": "0", "metar": "METAR KTYQ 160415Z AUTO 02005KT 10SM CLR 18/16 A3007
RMK AO2 T01820164" }, { "date": { "pretty": "12:35 AM EDT on July 16, 2016", "year": "2016",
"mon": "07", "mday": "16", "hour": "00", "min": "35", "tzname": "America/Indianapolis" },
"utcdate": { "pretty": "4:35 AM GMT on July 16, 2016", "year": "2016", "mon": "07", "mday":
"16", "hour": "04", "min": "35", "tzname": "UTC" }, "tempm": "17.7", "tempi": "63.9",
"dewptm": "16.3", "dewpti": "61.3", "hum": "91", "wspdm": "7.4", "wspdi": "4.6", "wgustm": "-
9999.0", "wgusti": "-9999.0", "wdird": "10", "wdire": "North", "vism": "16.1", "visi": "10.0",
```

```
"pressurem": "1018.2", "pressurei": "30.07", "windchillm": "-999", "windchilli": "-999",
"heatindexm": "-9999", "heatindexi": "-9999", "precipm": "-9999.00", "precipi": "-9999.00",
"conds": "Clear", "icon": "clear", "fog": "0", "rain": "0", "snow": "0", "hail": "0",
"thunder": "0", "tornado": "0", "metar": "METAR KTYQ 160435Z AUTO 01004KT 10SM CLR 18/16 A3007
RMK AO2 T01770163" } } }
```

Questo non è ciò che considereresti dati altamente leggibili. Questo è facilmente risolvibile impostando l'impostazione di una singola proprietà sul JsonFormatter predefinito in App_Start / ApiConfig.cs:

```
    // Either one of these will format your JSON in a readable format.  Setting both provides
no additional benefit.
    config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
MediaTypeHeaderValue("text/html"));
    // OR
    config.Formatters.JsonFormatter.Indent = true;
```

Leggi Configura un'applicazione API Web per rispondere con dati JSON belli / formattati di defualt online: https://riptutorial.com/it/asp-net-web-api/topic/6682/configura-un-applicazione-api-web-per-rispondere-con-dati-json-belli---formattati-di-defualt

Capitolo 7: Creazione di un ActionFilterAttribute personalizzato

introduzione

Filtri di azione Gli attributi sono una parte di ASP .NET Framework che ritengo utile per seguire il principio DRY. Puoi sostituire diverse linee di logica comune con un semplice tag dichiarativo. Il framework fornisce diversi utili attributi del filtro di azione per impostazione predefinita, come Autorizza e gestisce gli attributi di errore. Questa guida ha lo scopo di mostrarti come creare il tuo attributo personalizzato.

Examples

EnsurePresenseOfAttribute

Questo è un esempio di un attributo che ho creato per convalidare che i parametri richiesti sono stati assegnati nell'oggetto di richiesta ricevuto in una rotta POST. Ho deciso questo approccio perché l'approccio standard ModelState.IsValid non era valido. Questo perché gli attributi richiesti variano in base all'azione che viene chiamata.

// ATTRIBUISCE SOLO VALIDO PER I METODI [AttributeUsage (AttributeTargets.Method)] // INHERIT ActionFilterAttribute public class EnsurePresencesOfAttribute: ActionFilterAttribute {// ReSharper disabilita una volta IncoStringName stringa pubblica richiesta {get; impostato; }

```
// VALIDATE REQUIRED ATTRIBUTES
// FOR NON-ASYNC REQUESTS
public override void OnActionExecuting(HttpActionContext context)
{
    Dictionary<string, object> model = context.ActionArguments;
    var serialstring = JsonConvert.SerializeObject(model);
    foreach (var requirement in required.Split(','))
    {
        if (serialstring.Contains($"{requirement}\":null"))
        {
            ValueError(context, requirement);
            return;
        }
    }
    base.OnActionExecuting(context);
}

// VALIDATE THE REQUIRED ATTRIBUTES ARE PRESENT
// FOR ASYNC REQUESTS
public override Task OnActionExecutingAsync(HttpActionContext context, CancellationToken
token)
{
    Dictionary<string, object> model = context.ActionArguments;
    var serialstring = JsonConvert.SerializeObject(model);
    foreach (var requirement in required.Split(','))
```

```
    {
        if (serialstring.Contains($"{requirement}\":null"))
        {
            ValueError(context, requirement);
            return Task.FromResult(0);
        }
    }
    return base.OnActionExecutingAsync(context, token);
}

// LOG ERROR AND RETURN AND SET ERROR RESPONSE
private static void ValueError(HttpActionContext context, string requirement)
{
    var action = context.ActionDescriptor.ActionName;
    AppUtils.LogError($"{action} Failed : Missing Required Attribute {requirement}. ");
    using (var controller = new BaseApiController { Request = new HttpRequestMessage() })
    {
        controller.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey, new
HttpConfiguration());
        context.Response = controller.InvalidInputResponse();
    }
}
```

Controller prima di EnsuresPresenseOf Attribute

```
        [HttpPost]
        [Route("api/Fitbit/Activity/Stats")]
        public async Task<HttpResponseMessage> ActivityStats(FitbitRequestDTO request)
        {
            if (string.IsNullOrEmpty(request.PatientId) ||
string.IsNullOrEmpty(request.DeviceId))
                return InvalidInputResponse();
            try
            {
                var tokenErrorResponse = await EnsureToken(request);
                if (tokenErrorResponse != null)
                    return tokenErrorResponse;
                var client = GetFitbitClient();
                var stats = await client.GetActivitiesStatsAsync();
                return OkResponse(stats);
            }
            catch (Exception e)
            {
                const string function = " ActivityStats ";
                AppUtils.LogException(function, e);
                return SystemErrorResponse(function, e);
            }
        }
```

Controller di aggiornamento

```
        public async Task<HttpResponseMessage> ActivityStats(FitbitRequestDTO request)
        {
            try
            {
                var tokenErrorResponse = await EnsureToken(request);
                if (tokenErrorResponse != null)
```

```
                return tokenErrorResponse;
            var client = GetFitbitClient();
            var stats = await client.GetActivitiesStatsAsync();
            return OkResponse(stats);
        }
        catch (Exception e)
        {
            const string function = " ActivityStats ";
            AppUtils.LogException(function, e);
            return SystemErrorResponse(function, e);
        }
    }
```

Leggi Creazione di un ActionFilterAttribute personalizzato online: https://riptutorial.com/it/asp-net-web-api/topic/8989/creazione-di-un-actionfilterattribute-personalizzato

Capitolo 8: Instradamento degli attributi in WebAPI

introduzione

Come suggerisce il nome, questo utilizza gli attributi per il percorso. Ciò fornisce all'utente un maggiore controllo sull'URI nella WebAPI. Ad esempio, è possibile descrivere le gerarchie della risorsa. Tuttavia, il precedente "routing convenzionale" è pienamente supportato. Gli utenti possono avere una combinazione di entrambi.

Sintassi

- [RoutePrefix ("api / books")] - per la classe controller
- [Route ("getById")] - per le azioni
- [Route ("~ / api / authors / {authorId: int} / books")] - per il prefisso del percorso prioritario

Parametri

Nome del parametro	Dettagli
RoutePrefix	attributo alla classe controller. tutti i prefissi di url comuni nelle azioni sono bastonati qui. prende la stringa come input
Itinerario	attributo alle azioni del controller. ogni azione avrà una rotta assunta con (non necessariamente)
Rotta ("~ / api /")	questo sovrascrive il prefisso del percorso

Osservazioni

Attualmente, Attribute Routes non ha `Controller specific Message Handlers` . Poiché non è possibile specificare quale gestore eseguire per quale percorso al momento della dichiarazione. Questo è possibile nel `Conventional Routing` .

Examples

Routing degli attributi di base

Basta aggiungere un attributo all'azione del controller

```
[Route("product/{productId}/customer")]
```

```
public IQueryable<Product> GetProductsByCustomer(int productId)
{
    //action code goes here
}
```

questo verrà interrogato come `/product/1/customer` e `productId=1` verrà inviato all'azione del controller.

Assicurati che quello all'interno di "{}" e il parametro di azione siano gli stessi. `productId` in questo caso.

prima di usare questo, devi specificare che stai usando l'instradamento degli attributi per:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
    }
}
```

Attributo prefisso di percorso

Nei casi in cui è necessaria una porzione comune della rotta per tutte le rotte all'interno di un controller, viene utilizzato l'attributo `RoutePrefix` .

Nell'esempio seguente, la parte di codice api / studenti è comune e quindi è possibile definire `RoutePrefix` ed evitare di utilizzarlo ripetutamente.

```
[RoutePrefix("api/students")]
public class StudentController : ApiController
{
[Route("")]
public IEnumerable<Student> Get()
{
    //action code goes here
}

[Route("{id:int}")]
public Student Get(int id)
{
    //action code goes here
}

[Route("")]
public HttpResponseMessage Post(Student student)
{
    //action code goes here
}

}
```

Leggi Instradamento degli attributi in WebAPI online: https://riptutorial.com/it/asp-net-web-api/topic/9440/instradamento-degli-attributi-in-webapi

Capitolo 9: Negoziazione del contenuto dell'API Web ASP.NET

Examples

Informazioni di base sulla negoziazione del contenuto dell'API Web ASP.NET

La negoziazione del contenuto può essere definita come il processo di selezione della migliore rappresentazione per una determinata risorsa. Quindi la negoziazione del contenuto significa che il client e il server possono negoziare tra loro in modo che il cliente possa ottenere i dati in base al formato richiesto.

Ci sono tre punti da cui dipende internet,

- La risorsa
- Un puntatore a risorsa (URL)
- Rappresentazione di risorse

Il terzo punto è più importante degli altri due, perché tutto funziona sulla base di come possiamo vedere la risorsa. Possiamo rappresentare una risorsa in due formati.

1. Formato XML (Extensible Markup Language)
2. Formato JSON (notazione oggetto JavaScript)

Uno degli standard del servizio RESTful è che, il client dovrebbe avere la capacità di decidere in quale formato vogliono la risposta sia in JSON che in XML. Una richiesta inviata al server include un'intestazione Accept. Utilizzando l'intestazione Accept il client può specificare il formato per la risposta.

Per esempio,

`Accept: application/xml` risultati restituiti da `Accept: application/xml` in formato XML

`Accept: application/json` restituisce il risultato in formato JSON

A seconda del valore di intestazione Accept nella richiesta, il server invia la risposta. Questo è chiamato Content Negotiation.

Cosa succede dietro la scena quando richiediamo dati in formato specifico?

Il controller API Web ASP.NET genera i dati che vogliamo inviare al client e consegna i dati alla pipeline dell'API Web che cerca l'intestazione Accept del client. Quindi, scegliere un formattatore appropriato per formattare i dati.

Poiché l'API Web ASP.NET è notevolmente estensibile, possiamo anche specificare più valori per l'intestazione di accettazione nell'intestazione della richiesta.

```
Accept: application/xml,application/json
```

Nel caso precedente, il server sceglie il primo formattatore per formattare i dati di risposta.

Possiamo anche specificare il fattore di qualità nell'intestazione di accettazione. In questo caso, il server sceglie un formato con un fattore di qualità più elevato.

```
Accept: application/json;q=0.8,application/xml;q=0.5
```

Se non si specifica alcuna intestazione Accept, il server predefinito sceglie il formattatore JSON.

Quando la risposta viene inviata al client nel formato richiesto, si noti che l'intestazione `Content-Type` della risposta è impostata sul valore appropriato. Ad esempio, se il client ha richiesto `application/xml` , il server invia i dati in formato XML e imposta anche `Content-Type=application/xml` .

I formattatori vengono utilizzati dal server per entrambi i messaggi di richiesta e risposta. Quando il client invia una richiesta al server, impostiamo l'intestazione Content-Type sul valore appropriato per consentire al server di conoscere il formato dei dati che stiamo inviando.

Ad esempio, se il client invia dati JSON, l'intestazione Content-Type è impostata su `application/json` . Il server sa che ha a che fare con i dati JSON, quindi usa JSON formattatter per convertire i dati JSON in .NET Type. Analogamente, quando una risposta viene inviata dal server al client, a seconda del valore di intestazione Accept, viene utilizzato il formattatore appropriato per convertire il tipo .NET in JSON, XML, ecc.

Esempio di diversi tipi di formato di risposta:

application / json:

```
{
  "Email": "sample string 1",
  "HasRegistered": true,
  "LoginProvider": "sample string 3"
}
```

application / xml:

```
<UserInfoViewModel xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/WebApiDemo.Models">
  <Email>sample string 1</Email>
  <HasRegistered>true</HasRegistered>
  <LoginProvider>sample string 3</LoginProvider>
</UserInfoViewModel>
```

Le moderne applicazioni web possono fornire dati in varie lingue e formati. Pertanto, se sviluppiamo la nostra API per coprire gli utenti globali in tutto il mondo, la Negoziazione dei contenuti è rilevante.

Negoziazione del contenuto nell'API Web

Capire il concetto

Per comprendere la negoziazione del contenuto nell'API Web, è importante comprendere il termine `Resource` .

Sul web, qualsiasi informazione a cui possiamo accedere può essere indicata come `HTTP resource` . C'è una quantità enorme di materiale da visualizzare sul web che ha diversi tipi di contenuti come documenti html, immagini, video, audio, file eseguibili, fogli di calcolo, ecc. Possiamo ottenere qualsiasi risorsa facendo una richiesta http alla risorsa uri. La risposta http per la richiesta, restituisce la risorsa e specifica anche il tipo di contenuto, che è `also known as media type` .

Per accedere a qualsiasi risorsa, il cliente può effettuare una richiesta http fornendo uri specifici della risorsa e i verbi http. Tuttavia, in aggiunta a questo, il client può anche specificare il tipo di accettazione che è il formato del contenuto che l'utente sta cercando. Il "tipo di accettazione" può essere definito nelle intestazioni delle richieste http come intestazione `"accept"` .

Il server quindi controlla l'intestazione "accetta" dalle richieste e restituisce la risposta nel formato specificato, se disponibile. Si noti che il server può restituire la risposta nella rappresentazione richiesta solo **se è disponibile** . Se la rappresentazione richiesta non è disponibile, restituisce la risorsa nella rappresentazione predefinita. Questo è il motivo per cui è chiamato negoziazione del contenuto.

Un esempio pratico

Ad esempio, supponiamo che tu stia facendo una richiesta a http://example.com/customer/1 per ottenere le informazioni del cliente con l'id 1. Se non specifichi l'intestazione "accept" nella richiesta, il il server restituirà la rappresentazione predefinita di questa risorsa.

Supponiamo che il server possa restituire `json and xml both` le informazioni sul cliente in `json and xml both` . Ora, è sul client per specificare il formato richiesto delle informazioni sul cliente nell'intestazione "accetta" nella richiesta. Il valore dell'intestazione `"accept"` può essere `"application/json"` per la rappresentazione json, o `"text/xml"` per la rappresentazione xml. Il server restituirà quindi la risposta secondo il formato richiesto.

Se il formato richiesto è "text / html" che non è supportato da questo host (come in questo esempio), *`simply return the resource in the default format`* . La risposta http contiene un'intestazione `"content-type"` che indica al client il formato della risorsa.

Si noti che anche nel caso in cui la rappresentazione richiesta della risorsa non sia disponibile, viene comunque restituita la rappresentazione predefinita della risorsa.

Questo è il motivo per cui viene indicato come negoziazione del contenuto .

> Il client negozia la rappresentazione della risposta, tuttavia, se non è disponibile, ottiene quella predefinita.

Come configurare in Web API

Nell'API Web, la negoziazione del contenuto può essere configurata nella classe `WebAPIConfig` come

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

È anche possibile sovrascrivere la negoziazione del contenuto predefinita nell'API Web implementando l'interfaccia IContentNegotiator e il relativo metodo Negotiate, quindi configurarlo nella riga della pipe di richiesta dell'API Web, nel file WebAPI.config come indicato di seguito:

```
GlobalConfiguration.Configuration.Services.Replace(typeof(IContentNegotiator), new
CustomContentNegotiator());
```

Di seguito è riportato un esempio di implementazione del metodo Negotiate.

```
public class CustomContentNegotiator : DefaultContentNegotiator
    {
        public override ContentNegotiationResult Negotiate(Type type, HttpRequestMessage
request, IEnumerable<MediaTypeFormatter> formatters)
        {
            var result = new ContentNegotiationResult(new JsonMediaTypeFormatter(), new
MediaTypeHeaderValue("application/json"));
            return result;
        }
```

Leggi Negoziazione del contenuto dell'API Web ASP.NET online: https://riptutorial.com/it/asp-net-web-api/topic/7654/negoziazione-del-contenuto-dell-api-web-asp-net

Capitolo 10: OData con Asp.net Web API

Examples

Installa i pacchetti OData

Dal menu Strumenti, selezionare NuGet Package Manager> Console Gestione pacchetti. Nella finestra della console di Package Manager, digitare:

```
Install-Package Microsoft.AspNet.Odata
```

Questo comando installa i pacchetti OData NuGet più recenti.

Abilita Entity Framework

Per questo tutorial, utilizzeremo il codice Entity Framework (EF) First per creare il database back-end.

> Web API OData non richiede EF. Utilizzare qualsiasi livello di accesso ai dati che possa convertire le entità del database in modelli.

Innanzitutto, installa il pacchetto NuGet per EF. Dal menu **Strumenti** , selezionare **NuGet Package Manager** > **Console Gestione pacchetti** . Nella finestra della console di Package Manager, digitare:

```
Install-Package EntityFramework
```

Aprire il file Web.config e aggiungere la seguente sezione all'interno dell'elemento di **configurazione** , dopo l'elemento **configSections** .

```
<configuration>
  <configSections>
    <!-- ... -->
  </configSections>

  <!-- Add this: -->
  <connectionStrings>
    <add name="ProductsContext" connectionString="Data Source=(localdb)\v11.0;
        Initial Catalog=ProductsContext; Integrated Security=True;
MultipleActiveResultSets=True;
        AttachDbFilename=|DataDirectory|ProductsContext.mdf"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Questa impostazione aggiunge una stringa di connessione per un database LocalDB. Questo database verrà utilizzato quando si esegue l'app localmente.

Quindi, aggiungi una classe denominata **ProductsContext** alla cartella Modelli:

```
using System.Data.Entity;
namespace ProductService.Models
{
    public class ProductsContext : DbContext
    {
        public ProductsContext()
                : base("name=ProductsContext")
        {
        }
        public DbSet<Product> Products { get; set; }
    }
}
```

Nel costruttore, **"name = ProductsContext"** fornisce il nome della stringa di connessione.

Configura l'endpoint OData

Apri il file App_Start / WebApiConfig.cs. Aggiungi le seguenti affermazioni **usando** :

```
using ProductService.Models;
using System.Web.OData.Builder;
using System.Web.OData.Extensions;
```

Quindi aggiungere il seguente codice al metodo **Register** :

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // New code:
        ODataModelBuilder builder = new ODataConventionModelBuilder();
        builder.EntitySet<Product>("Products");
        config.MapODataServiceRoute(
            routeName: "ODataRoute",
            routePrefix: null,
            model: builder.GetEdmModel());
    }
}
```

Questo codice fa due cose:

- Crea un Entity Data Model (EDM).
- Aggiunge un percorso.

Un EDM è un modello astratto dei dati. L'EDM viene utilizzato per creare il documento dei metadati del servizio. La classe **ODataConventionModelBuilder** crea un EDM utilizzando le convenzioni di denominazione predefinite. Questo approccio richiede il codice minimo. Se si desidera un maggiore controllo **sull'EDM** , è possibile utilizzare la classe **ODataModelBuilder** per creare l'EDM aggiungendo proprietà, chiavi e proprietà di navigazione in modo esplicito.

Una rotta indica all'API Web come indirizzare le richieste HTTP all'endpoint. Per creare una route OData v4, chiamare il metodo di estensione **MapODataServiceRoute** .

Se l'applicazione ha più endpoint OData, creare un percorso separato per ciascuno. Assegnare a ogni percorso un nome e un prefisso univoci.

Aggiungi il controller OData

Un controller è una classe che gestisce le richieste HTTP. Crei un controller separato per ogni entità impostata nel tuo servizio OData. In questo tutorial, creerai un controller, per l'entità Product.

In Esplora soluzioni, fare clic con il pulsante destro del mouse sulla cartella Controller e selezionare **Aggiungi** > **Classe** . Assegna un nome alla classe ProductsController.

> La versione di questo tutorial per OData v3 utilizza lo scaffold Add Controller. Attualmente, non esiste un'impalcatura per OData v4.

Sostituire il codice boilerplate in ProductsController.cs con quanto segue.

```
using ProductService.Models;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.OData;
namespace ProductService.Controllers
{
    public class ProductsController : ODataController
    {
        ProductsContext db = new ProductsContext();
        private bool ProductExists(int key)
        {
            return db.Products.Any(p => p.Id == key);
        }
        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

Il controller utilizza la classe **ProductsContext** per accedere al database utilizzando EF. Si noti che il controller sovrascrive il metodo **Dispose** per eliminare **ProductsContext** .

Questo è il punto di partenza per il controller. Successivamente, aggiungeremo metodi per tutte le operazioni CRUD.

Esecuzione di CRUD sul set di entità

Interrogazione del set di entità

Aggiungere i seguenti metodi a **ProductsController** .

```
[EnableQuery]
public IQueryable<Product> Get()
{
    return db.Products;
}

[EnableQuery]
public SingleResult<Product> Get([FromODataUri] int key)
{
    IQueryable<Product> result = db.Products.Where(p => p.Id == key);
    return SingleResult.Create(result);
}
```

La versione senza parametri del metodo Get restituisce l'intera raccolta Prodotti. Il metodo Get con un parametro chiave cerca un prodotto tramite la sua chiave (in questo caso, la proprietà Id).

L'attributo **[EnableQuery]** consente ai client di modificare la query, utilizzando opzioni di query quali $ filter, $ sort e $ page. Per ulteriori informazioni, vedere Supporto delle opzioni di query OData .

Aggiunta di una entità al set di entità

Per consentire ai client di aggiungere un nuovo prodotto al database, aggiungere il seguente metodo a **ProductsController** .

```
public async Task<IHttpActionResult> Post(Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    db.Products.Add(product);
    await db.SaveChangesAsync();
    return Created(product);
}
```

Aggiornamento di un'entità

OData supporta due semantiche diverse per l'aggiornamento di un'entità, PATCH e PUT.

- PATCH esegue un aggiornamento parziale. Il client specifica solo le proprietà da aggiornare.
- PUT sostituisce l'intera entità.

Lo svantaggio di PUT è che il client deve inviare valori per tutte le proprietà nell'entità, inclusi i valori che non cambiano. Le specifiche OData indicano che PATCH è preferito.

In ogni caso, ecco il codice per entrambi i metodi PATCH e PUT:

```
public async Task<IHttpActionResult> Patch([FromODataUri] int key, Delta<Product> product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    var entity = await db.Products.FindAsync(key);
    if (entity == null)
    {
        return NotFound();
    }
    product.Patch(entity);
    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return Updated(entity);
}

public async Task<IHttpActionResult> Put([FromODataUri] int key, Product update)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    if (key != update.Id)
    {
        return BadRequest();
    }
    db.Entry(update).State = EntityState.Modified;
    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return Updated(update);
}
```

Nel caso di PATCH, il controller usa il tipo **Delta <T>** per tracciare le modifiche.

Eliminazione di un'entità

Per consentire ai client di eliminare un prodotto dal database, aggiungere il seguente metodo a **ProductsController** .

```
public async Task<IHttpActionResult> Delete([FromODataUri] int key)
{
    var product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }
    db.Products.Remove(product);
    await db.SaveChangesAsync();
    return StatusCode(HttpStatusCode.NoContent);
}
```

Leggi OData con Asp.net Web API online: https://riptutorial.com/it/asp-net-web-api/topic/6019/odata-con-asp-net-web-api

Capitolo 11: Routing dell'URL API Web

Examples

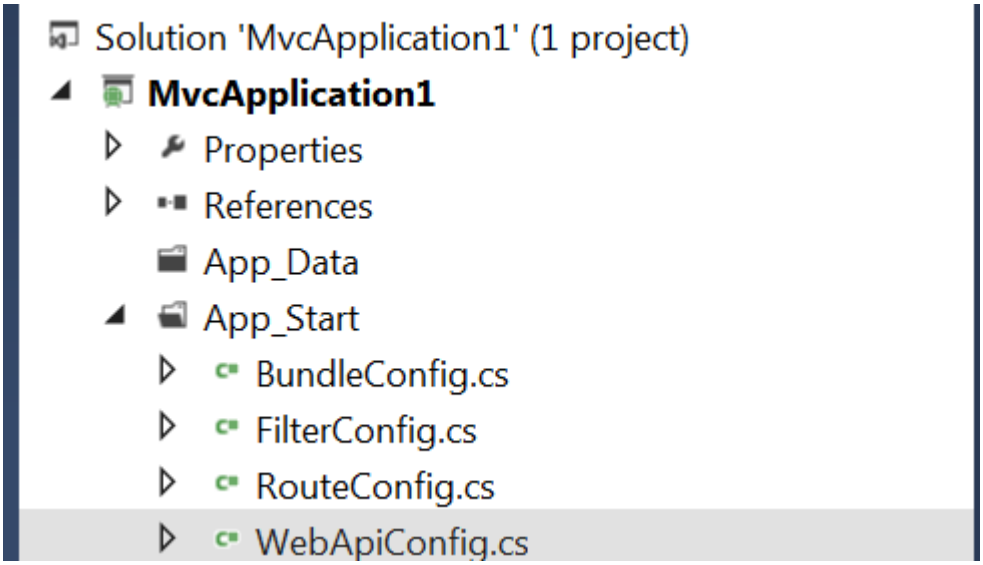
Come funziona il routing in asp.net webapi

Nell'API Web ASP.NET, un controller è una classe che gestisce le richieste HTTP. I metodi pubblici del controller sono chiamati metodi di azione o semplicemente azioni.

Quando il framework API Web riceve una richiesta, indirizza la richiesta a un'azione. Per determinare quale azione invocare, il framework utilizza una tabella di routing. Il modello di progetto di Visual Studio per Web API crea una route predefinita:

```
routes.MapHttpRoute(
    name: "API Default",
    routeTemplate: "**api/{controller}/{id}**",
    defaults: new { id = RouteParameter.Optional }
);
```

Questa route è definita nel file WebApiConfig.cs, che viene inserito nella directory App_Start:



Ogni voce nella tabella di routing contiene un modello di percorso. Il modello di route predefinito per l'API Web è " **api / {controller} / {id}** ". In questo modello, " **api** " è un segmento di percorso letterale e { **controller** } e { **id** } sono variabili segnaposto.

Quando il framework API Web riceve una richiesta HTTP, prova a far corrispondere l'URI con uno dei modelli di instradamento nella tabella di routing. Se nessuna route corrisponde, il client riceve un errore 404.

Ad esempio, i seguenti URI corrispondono alla route predefinita:

- / API / valori
- / Api / valori / 1

Tuttavia, il seguente URI non corrisponde, perché manca il segmento " **api** ":

- / Valori / 1

Una volta trovata una rotta corrispondente, l'API Web seleziona il controller e l'azione:

- Per trovare il controller, l'API Web aggiunge "Controller" al valore della variabile {controller}.
- Per trovare l'azione, l'API Web esamina il metodo HTTP e quindi cerca un'azione il cui nome inizia con il nome del metodo HTTP. Ad esempio, con una richiesta GET, l'API Web cerca un'azione che inizia con "Get ...", come "GetEmployee" o "GetAllEmployees". Questa convenzione si applica solo ai metodi GET, POST, PUT e DELETE.

È possibile abilitare altri metodi HTTP utilizzando gli attributi sul controller. Ne vedremo un esempio dopo.

- Altre variabili segnaposto nel modello di percorso, come {id}, sono mappate ai parametri di azione.

Metodi HTTP Invece di utilizzare la convenzione di denominazione per i metodi HTTP, è possibile specificare esplicitamente il metodo HTTP per un'azione decorando il metodo di azione con l'attributo HttpGet, HttpPut, HttpPost o HttpDelete.

Nell'esempio seguente, il metodo EmployeeGetEmployee viene associato alle richieste GET:

```
 public class EmployeesController : ApiController
    {
        [HttpGet]
        public EmployeeGetEmployee(id) {}
    }
```

Per consentire più metodi HTTP per un'azione o per consentire metodi HTTP diversi da GET, PUT, POST e DELETE, utilizzare l'attributo AcceptVerbs, che accetta un elenco di metodi HTTP.

```
public class EmployeesController: ApiController
{
    [AcceptVerbs("GET", "HEAD")]
    public Employee GetEmployee (id) { }
}
```

Routing per nome dell'azione

Con il modello di instradamento predefinito, l'API Web utilizza il metodo HTTP per selezionare l'azione. Tuttavia, puoi anche creare una rotta in cui il nome dell'azione è incluso nell'URI:

```
routes.MapHttpRoute(
    name: "ActionApi",
    routeTemplate: "api/{controller}/{action}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

In questo modello di percorso, il parametro {action} indica il metodo di azione sul controller. Con

questo stile di routing, utilizzare gli attributi per specificare i metodi HTTP consentiti. Ad esempio, supponiamo che il controller abbia il seguente metodo:

```
public class EmployeesController: ApiController
{
    [HttpGet]
    public List<Employee> GetAllEmployees();
}
```

In questo caso, una richiesta GET per " **api / Employees / GetAllEmployees** " si assocerebbe al metodo GetAllEmployees.

Puoi sovrascrivere il nome dell'azione usando l'attributo ActionName. Nell'esempio seguente, esistono due azioni che si **associano** a " **api / Employees / ShowAllEmployees / id** . Uno supporta GET e l'altro supporta POST:

```
public class EmployeesController : ApiController
{
    [HttpGet]
    [ActionName("ShowAllEmployees")]
    public List<Employee> GetAll(int id);

    [HttpPost]
    [ActionName("ShowAllEmployees")]
    public void GetAll (int id);

}
```

Non-azioni

È possibile impedire che un metodo venga richiamato come azione utilizzando l'attributo NonAction. Questo segnala al framework che il metodo non è un'azione, anche se altrimenti corrisponderebbe alle regole di routing.

```
 [NonAction]
public string GetValues() { ... }
```

Esempi di routing basati sui verbi.

Lo stesso URl per diversi metodi http agisce in modo diverso. Di seguito è riportata una tabella raffigurante la stessa.

VERBO HTTP	URL	DESCRIZIONE
OTTENERE	/ API / studenti	Restituisce tutti gli studenti
OTTENERE	/ API / studenti / 5	Restituisce i dettagli dell'ID studente = 5
INVIARE	/ API / studenti	Aggiungi un nuovo studente
METTERE	/ API / studenti / 5	Aggiorna studente con Id = 5

VERBO HTTP	URL	DESCRIZIONE
ELIMINA	/ API / studenti / 5	Elimina studente con Id = 5

Leggi Routing dell'URL API Web online: https://riptutorial.com/it/asp-net-web-api/topic/2432/routing-dell-url-api-web

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con asp.net-web-api	Arif, BehrouzMoslem, Community, The_Outsider
2	Abilitazione di CORS API WEB ASP.NET	Alexei, Oluwafemi
3	API Web ASP.NET MediaTypeFormatter	Keyur Ramoliya
4	Avvio rapido: lavorare con JSON	Brett Veenstra
5	caching	Sibeesh Venu
6	Configura un'applicazione API Web per rispondere con dati JSON belli / formattati di defualt	Patrick
7	Creazione di un ActionFilterAttribute personalizzato	Antarr Byrd, Blanthor
8	Instradamento degli attributi in WebAPI	Ajay Aradhya, The_Outsider
9	Negoziazione del contenuto dell'API Web ASP.NET	Amit Shahani, Keyur Ramoliya
10	OData con Asp.net Web API	Yushell
11	Routing dell'URL API Web	ravindra, riteshmeher, The_Outsider