eBook Gratuit

APPRENEZ asp.net-web-api

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#asp.net-web-api

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: asp-net-web-api

It is an unofficial and free asp.net-web-api ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official asp.net-web-api.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec asp.net-web-api

Remarques

Cette section donne un aperçu de ce qu'est asp.net-web-api et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans asp.net-web-api, et établir un lien vers les sujets connexes. La documentation de asp.net-web-api étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

Installation ou configuration

Instructions détaillées sur l'installation ou l'installation de asp.net-web-api.

Quoi et pourquoi API Web ASP.NET?

Quelle? : Un framework totalement supporté et extensible pour la construction de noeuds finaux basés sur HTTP. Dans le monde de HTML5, les appareils mobiles et les techniques de développement modernes HTTP sont devenus l'option par défaut pour la création de services riches et évolutifs. L'API Web ASP.NET fournit un ensemble d'options par défaut facile à utiliser, mais fournit également une infrastructure d'extensibilité poussée pour répondre aux exigences de tout scénario utilisant HTTP.

Pourquoi? :

- Une application HTML5 nécessitant une couche de services.
- Une application mobile qui nécessite une couche de services.
- Une application de bureau client-serveur qui nécessite une couche de services.

Pour ajouter une API Web à une application MVC existante.

Utilisez Nuget pour rechercher le package Web Api.

Vous pouvez le faire soit en utilisant les packages de gestion de Nuget et en recherchant le package Web Api, soit en utilisant Nuget Package Manager et en saisissant

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

Ajouter WebApiConfig.cs au dossier App_Start / Le fichier de configuration doit contenir ceci.

```
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 }
            );
        });
    }
 }
}
```

Source: Configuration de l'API Web ASP.NET

Ajouter `GlobalConfiguration.Configure(WebApiConfig.Register);` dans Application_Start du fichier Global.asax.

Lire Démarrer avec asp.net-web-api en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/1058/demarrer-avec-asp-net-web-api

Chapitre 2: ASP.NET WEB API Activer CORS

Examples

Activation de CORS pour 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...
}
```

Activation globale de CORS pour Web API 2

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

Activation de CORS dans Asp.Net 5 pour tous les domaines et méthodes

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

    // ...
}

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

    // ...
}
```

Activation de CORS dans Asp.Net 5 pour des domaines et des méthodes spécifiques

```
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")));
}
```

Configurez CORS pour WebAPI 2 avec l'authentification Windows

La configuration côté serveur suivante permet à la requête CORS de fonctionner avec l'authentification Windows (aucun anonyme ne doit être activé dans IIS).

web.config - Autorise les demandes de contrôle en amont non authentifiées (anonymes) (OPTIONS)

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

global.asax.cs - répond correctement avec les en-têtes qui permettent à l'appelant d'un autre domaine de recevoir des données

```
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();
    }
}
```

CORS permettant

```
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);
}
```

Envoyer correctement la demande authentifiée à partir de jQuery sur le noeud final Web API 2

L'exemple suivant montre comment construire correctement les requêtes GET et POST sur Web API 2 (CORS doit être configuré côté serveur, s'il est envoyé depuis un autre domaine):

```
<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>
```

Code côté serveur:

```
    [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;
    }
```

Envoyer correctement la demande authentifiée à partir d'AngularJS contre le point de terminaison de l'API Web 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>
```

Lire ASP.NET WEB API Activer CORS en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/4185/asp-net-web-api-activer-cors

Chapitre 3: ASP.NET Web API MediaTypeFormatter

Examples

MediaTypeFormatter Informations de base

`MediaTypeFormatter` est une classe abstraite à partir de laquelle les classes `JsonMediaTypeFormatter` et `XmlMediaTypeFormatter` héritent. Ici, la classe `JsonMediaTypeFormatter` gère les objets JSON et la classe `XmlMediaTypeFormatter` gère les objets XML.

Renvoie uniquement JSON, quelle que soit la valeur de l'en-tête Accept:

Pour renvoyer uniquement les objets JSON dans la réponse à la demande de temps, cliquez sur Accepter la valeur d'en-tête de la demande si `application/json` ou `application/xml` écrit la ligne suivante dans la méthode `Register` de la classe `WebApiConfig` .

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

Ici, `config` est un objet de la classe `HttpConfiguration` . Cette ligne de code supprime complètement `XmlFormatter` ce qui oblige l'API Web ASP.NET à toujours renvoyer JSON, quelle que soit la valeur de l'en-tête Accept dans la demande du client. Utilisez cette technique lorsque vous souhaitez que votre service ne prenne en charge que JSON et non XML.

Renvoie uniquement le code XML indépendamment de la valeur de l'en-tête Accept:

Pour renvoyer uniquement les objets XML dans la réponse à la demande météo, cliquez sur Accepter la valeur d'en-tête de la demande si `application/json` ou `application/xml` écrit la ligne suivante dans la méthode `Register` de la classe `WebApiConfig` .

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

Ici, `config` est un objet de la classe `HttpConfiguration` comme décrit ci-dessus. Cette ligne de code supprime complètement `JsonFormatter` ce qui oblige l'API Web ASP.NET à toujours renvoyer du code XML, quelle que soit la valeur de l'en-tête Accept dans la demande du client. Utilisez cette technique lorsque vous souhaitez que votre service ne prenne en charge que XML et non JSON.

Retourne JSON au lieu de XML:

1. Lorsqu'une demande est émise par le navigateur, le service API Web doit renvoyer JSON au lieu de XML.
2. Lorsqu'une requête est émise par un outil tel que le violon, la valeur de l'en-tête Accept doit être respectée. Cela signifie que si l'en-tête Accept est défini sur application / xml, le service doit renvoyer XML et s'il est défini sur application / json, le service doit renvoyer JSON.

Méthode 1:

Incluez la ligne suivante dans la méthode `Register` de la classe `WebApiConfig` .

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

Cela indique à l'API Web ASP.NET d'utiliser `JsonFormatter` lorsque la demande est faite pour `text/html` qui est la valeur par défaut pour la plupart des navigateurs. Le problème avec cette approche est que l'en `Content-Type` tête `Content-Type` de la réponse est défini sur `text/html` ce qui est trompeur.

Méthode 2:

Utilisez des formateurs personnalisés. Créez une classe dérivée de la classe `JsonMediaTypeFormatter` et implémentez la méthode `SetDefaultContentHeaders` .

Voici l'exemple de la classe de formatage JSON personnalisée qui renvoie le format JSON en réponse.

```
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");
    }
}
```

Et ceci est l'exemple du formateur de type Custom Media qui renvoie le format CSV en réponse.

```
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);
        }
```

```
    }
}
```

Après, l'implémentation de la classe de formatage personnalisée l'enregistre dans la méthode `Register` de la classe `WebApiConfig` .

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

Maintenant, selon votre formateur, vous obtiendrez une réponse et un `Content-Type` de `Content-Type` partir du serveur.

Lire ASP.NET Web API MediaTypeFormatter en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/7673/asp-net-web-api-mediatypeformatter

Chapitre 4: Configurez une application Web API pour répondre avec des données JSON jolies / formatées par défaut

Examples

Formatage JSON par défaut: efficacité au détriment de la lisibilité

Disons que vous avez un simple ApiController comme celui-ci:

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

        return obj;
    }
```

La représentation JSON résultante de cet objet ressemblera à ceci:

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

C'est probablement bien pour des réponses simples comme celle-ci, mais imaginez si vous avez un objet grand / complexe envoyé comme réponse:

```
"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" } } }
```

Ce n'est pas ce que vous considéreriez comme des données hautement lisibles. Cela est facilement résolu en définissant la définition d'une propriété unique sur le JsonFormatter par défaut dans 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;
```

Lire Configurez une application Web API pour répondre avec des données JSON jolies / formatées par défaut en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/6682/configurez-une-application-web-api-pour-repondre-avec-des-donnees-json-jolies---formatees-par-defaut

Chapitre 5: Création d'un ActionFilterAttribute personnalisé

Introduction

Filtres d'action Les attributs font partie du framework ASP .NET que je trouve utile pour suivre le principe DRY. Vous pouvez remplacer plusieurs lignes de logique commune par une simple balise déclarative. Le framework fournit par défaut plusieurs attributs de filtre d'action, tels que les attributs d'erreur Authorize et Handle. Ce guide est destiné à vous montrer comment créer votre propre attribut personnalisé.

Examples

EnsurePresenseOfAttribute

Voici un exemple d'attribut que j'ai créé pour valider que les paramètres requis ont été attribués dans l'objet de requête reçu dans un itinéraire POST. J'ai choisi cette approche car l'approche standard ModelState.IsValid n'était pas valide. En effet, les attributs requis varient en fonction de l'action appelée.

// ATTRIBUT UNIQUEMENT VALABLE POUR METHODS [AttributeUsage (AttributeTargets.Method)] // INHERIT ActionFilterAttribute classe publique EnsurePresencesOfAttribute: ActionFilterAttribute {/// ReSharper disable une fois InconsistentNaming public string required {get; ensemble; }

```
// 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();
    }
}
```

Contrôleur avant l'attribut EnsuresPresenseOf

```
        [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);
            }
        }
```

Mettre à jour le contrôleur

```
        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);
        }
    }
```

Lire Création d'un ActionFilterAttribute personnalisé en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/8989/creation-d-un-actionfilterattribute-personnalise

Chapitre 6: Démarrage rapide: Travailler avec JSON

Remarques

Exemples pour vous permettre de démarrer rapidement et correctement avec ASP.NET WebAPI

Examples

Renvoyer JSON depuis GET en utilisant des attributs

1. Configurez votre formateur et votre routage dans `Register` of (`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. Créer des méthodes dans 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 });

}
```

exécuter `GET /echo/foo`

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

exécuter `GET /echo/1241290805`

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

comme le framework de routage prend les conditions les plus spécifiques (type de données) lors du choix d'une méthode

Lire Démarrage rapide: Travailler avec JSON en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/3851/demarrage-rapide--travailler-avec-json

Chapitre 7: Mise en cache

Remarques

La mise en cache est le processus de stockage des données quelque part pour les demandes futures. Dans notre cas, nous pouvons éviter les accès indésirables à la base de données pour récupérer les données si nous les mettons en cache. .

Examples

System.Runtime.Caching (MemoryCache)

Importez l'espace de noms System.Runtime.Caching (Assurez-vous d'avoir ajouté la DLL System.Runtime.Caching à votre référence de projet).

Créez une instance de la classe MemoryCache.

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

Ajouter des valeurs à MemoryCache

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

Ici, «tag» est ma clé et «ca» est mes valeurs et DateTimeOffset.UtcNow.AddMinutes (5) sert à définir le cache pour cinq minutes à partir de maintenant.

Obtenir des valeurs de 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;
            }
```

Nous allons obtenir les valeurs de cache dans la variable res, rappelez-vous que ces valeurs ne seront là que pour cinq minutes. Vous pouvez toujours changer cela en fonction des besoins. Si la valeur n'est pas nulle, nous allons simplement la renvoyer et faire la manipulation et si elle est nulle, nous irons chercher les données de la base de données et ajouterons la valeur au cache.

Supprimer les valeurs de MemoryCache

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

Lire Mise en cache en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/7983/mise-en-cache

Chapitre 8: Négociation du contenu de l'API Web ASP.NET

Examples

Informations de base sur la négociation du contenu de l'API Web ASP.NET

La négociation de contenu peut être définie comme le processus de sélection de la meilleure représentation pour une ressource donnée. Ainsi, la négociation de contenu signifie que le client et le serveur peuvent négocier entre eux afin que le client puisse obtenir des données en fonction de leur format requis.

Internet dépend de trois points,

- La ressource
- Un pointeur vers la ressource (URL)
- Représentation de ressource

Le troisième point est plus important que les deux autres, car tout fonctionne sur la base de la manière dont nous pouvons voir la ressource. Nous pouvons représenter une ressource sous deux formats.

1. Format XML (Extensible Markup Language)
2. Format JSON (notation d'objet JavaScript)

L'un des standards du service RESTful est que le client doit avoir la possibilité de décider dans quel format il souhaite obtenir la réponse, en JSON ou XML. Une demande envoyée au serveur inclut un en-tête Accept. A l'aide de l'en-tête Accept, le client peut spécifier le format de la réponse.

Par exemple,

`Accept: application/xml` renvoie le résultat au format XML

`Accept: application/json` renvoie le résultat au format JSON

Selon la valeur d'en-tête Accept de la demande, le serveur envoie la réponse. Cela s'appelle la négociation de contenu.

Que se passe-t-il derrière la scène lorsque nous demandons des données dans un format spécifique?

Le contrôleur Web API ASP.NET génère les données que vous souhaitez envoyer au client et les transmet au pipeline d'API Web qui recherche ensuite l'en-tête Accept du client. Ensuite, choisissez un formateur approprié pour formater les données.

Comme l'API Web ASP.NET est très extensible, nous pouvons également spécifier plusieurs valeurs pour l'en-tête d'acceptation dans l'en-tête de la demande.

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

Dans le cas ci-dessus, le serveur choisit le premier formateur pour formater les données de la réponse.

Nous pouvons également spécifier le facteur de qualité dans l'en-tête accept. Dans ce cas, le serveur choisit un format présentant un facteur de qualité supérieur.

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

Si nous ne spécifions aucun en-tête Accept, le serveur choisit par défaut le formateur JSON.

Lorsque la réponse est envoyée au client dans le format demandé, notez que l'en `Content-Type` tête `Content-Type` de la réponse est défini sur la valeur appropriée. Par exemple, si le client a demandé l' `application/xml` , le serveur envoie les données au format XML et définit également `Content-Type=application/xml` .

Les formateurs sont utilisés par le serveur pour les messages de demande et de réponse. Lorsque le client envoie une demande au serveur, nous définissons l'en-tête Content-Type sur la valeur appropriée pour permettre au serveur de connaître le format des données que nous envoyons.

Par exemple, si le client envoie des données JSON, l'en-tête Content-Type est défini sur `application/json` . Le serveur sait qu'il traite les données JSON. Il utilise donc le formateur JSON pour convertir les données JSON en .NET Type. De même, lorsqu'une réponse est envoyée du serveur au client, en fonction de la valeur de l'en-tête Accept, le formateur approprié est utilisé pour convertir le type .NET en JSON, XML, etc.

Exemple de différents types de format de réponse:

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>
```

Les applications Web modernes peuvent fournir des données dans divers langages et formats. Ainsi, si nous développons notre API pour couvrir les utilisateurs mondiaux à travers le monde, la

négociation de contenu est pertinente.

Négociation de contenu dans l'API Web

Comprendre le concept

Pour comprendre la négociation de contenu dans l'API Web, il est important de comprendre le terme `Resource` .

Sur le Web, toute information à laquelle nous pouvons accéder peut être appelée `HTTP resource` . Il y a une quantité énorme de matériel à afficher sur le Web qui a différents types de contenu, tels que des documents HTML, des images, de la vidéo, de l'audio, des fichiers exécutables, des feuilles de calcul, etc. La réponse http pour la demande renvoie la ressource et spécifie également le type de contenu, `also known as media type` .

Pour accéder à une ressource, le client peut effectuer une requête http en fournissant une ressource spécifique uri et les verbes http. Cependant, en plus de cela, le client peut également spécifier le type d'acceptation, qui est le format du contenu recherché par l'utilisateur. Le type «accept-type» peut être défini dans les en-têtes de requête http comme en-tête `"accept"` .

Le serveur vérifie alors l'en-tête «accept» des demandes et renvoie la réponse au format spécifié, si disponible. Veuillez noter que le serveur ne peut renvoyer la réponse dans la représentation demandée que **si elle est disponible** . Si la représentation demandée n'est pas disponible, elle renvoie la ressource par défaut. C'est la raison pour laquelle cela s'appelle la négociation de contenu.

Un exemple pratique

Par exemple, supposons que vous faites une demande à http://example.com/customer/1 pour obtenir les informations du client avec l'ID 1. Si vous ne spécifiez pas l'en-tête «accept» dans la demande, le Le serveur retournera la représentation par défaut de cette ressource.

Supposons que le serveur puisse retourner les informations client dans `json and xml both` . Il appartient maintenant au client de spécifier le format requis des informations client dans l'en-tête «accept» de la demande. La valeur de l'en-tête `"accept"` peut être `"application/json"` pour la représentation json ou `"text/xml"` pour la représentation XML. Le serveur renverra alors la réponse selon le format demandé.

Si le format demandé est «text / html» qui n'est pas pris en charge par cet hôte (comme dans cet exemple), il *`simply return the resource in the default format`* . La réponse http contient un en-tête `"content-type"` qui indique au client le format de la ressource.

Veuillez noter que même dans le cas où la représentation demandée de la ressource n'est pas disponible, la représentation par défaut de la ressource est toujours renvoyée.

C'est pourquoi on parle de négociation de contenu .

Le client négocie la représentation de la réponse, cependant, s'il n'est pas disponible, il en obtient une par défaut.

Comment configurer dans l'API Web

Dans Web API, la négociation de contenu peut être configurée dans la classe `WebAPIConfig` tant que

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

Vous pouvez également remplacer la négociation de contenu par défaut dans l'API Web en implémentant l'interface IContentNegotiator et sa méthode Negotiate, puis la configurer dans la ligne de canal de demande d'API Web, dans le fichier WebAPI.config, comme suit:

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

Voici un exemple d'implémentation de la méthode 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;
        }
```

Lire Négociation du contenu de l'API Web ASP.NET en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/7654/negociation-du-contenu-de-l-api-web-asp-net

Chapitre 9: OData avec l'API Web Asp.net

Examples

Installer les packages OData

Dans le menu Outils, sélectionnez NuGet Package Manager> Console du gestionnaire de packages. Dans la fenêtre Console du Gestionnaire de packages, tapez:

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

Cette commande installe les derniers packages OData NuGet.

Activer le cadre d'entité

Pour ce tutoriel, nous utiliserons le code Entity Framework (EF) First pour créer la base de données principale.

> API Web OData ne nécessite pas EF. Utilisez toute couche d'accès aux données capable de traduire des entités de base de données en modèles.

Tout d'abord, installez le package NuGet pour EF. Dans le menu **Outils** , sélectionnez **NuGet Package Manager** > **Console du gestionnaire de packages** . Dans la fenêtre Console du Gestionnaire de packages, tapez:

```
Install-Package EntityFramework
```

Ouvrez le fichier Web.config et ajoutez la section suivante dans l'élément de **configuration** , après l'élément **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>
```

Ce paramètre ajoute une chaîne de connexion pour une base de données LocalDB. Cette base de données sera utilisée lorsque vous exécuterez l'application localement.

Ensuite, ajoutez une classe nommée **ProductsContext** au dossier Models:

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

Dans le constructeur, **"name = ProductsContext"** donne le nom de la chaîne de connexion.

Configurez le point de terminaison OData

Ouvrez le fichier App_Start / WebApiConfig.cs. Ajoutez les instructions suivantes en **utilisant** :

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

Ajoutez ensuite le code suivant à la méthode **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());
    }
}
```

Ce code fait deux choses:

- Crée un modèle de données d'entité (EDM).
- Ajoute un itinéraire.

Un EDM est un modèle abstrait des données. L'EDM est utilisé pour créer le document de métadonnées du service. La classe **ODataConventionModelBuilder** crée un **modèle** EDM en utilisant des conventions de dénomination par défaut. Cette approche nécessite le moins de code. Si vous souhaitez davantage de contrôle sur l'EDM, vous pouvez utiliser la classe **ODataModelBuilder** pour créer l'EDM en ajoutant explicitement des propriétés, des clés et des propriétés de navigation.

Une route indique à Web API comment acheminer les requêtes HTTP vers le noeud final. Pour créer un itinéraire OData v4, appelez la méthode d'extension **MapODataServiceRoute** .

Si votre application possède plusieurs points de terminaison OData, créez un itinéraire distinct pour chacun. Attribuez à chaque itinéraire un nom et un préfixe d'itinéraire uniques.

Ajouter le contrôleur OData

Un contrôleur est une classe qui gère les requêtes HTTP. Vous créez un contrôleur distinct pour chaque ensemble d'entités dans votre service OData. Dans ce tutoriel, vous créerez un contrôleur pour l'entité Produit.

Dans l'Explorateur de solutions, cliquez avec le bouton droit sur le dossier Contrôleurs et sélectionnez **Ajouter** > **Classe** . Nommez la classe ProductsController.

> La version de ce didacticiel pour OData v3 utilise l'échafaudage Add Controller. Actuellement, il n'y a pas d'échafaudage pour OData v4.

Remplacez le code passe-partout dans ProductsController.cs par le suivant.

```
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);
        }
    }
}
```

Le contrôleur utilise la classe **ProductsContext** pour accéder à la base de données à l'aide d'EF. Notez que le contrôleur remplace la méthode **Dispose** pour se débarrasser de **ProductsContext** .

C'est le point de départ du contrôleur. Ensuite, nous allons ajouter des méthodes pour toutes les opérations CRUD.

Exécution de CRUD sur l'ensemble d'entités

Interrogation de l'ensemble d'entités

Ajoutez les méthodes suivantes à **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 version sans paramètre de la méthode Get renvoie la collection entière de produits. La méthode Get avec un paramètre key recherche un produit par sa clé (dans ce cas, la propriété Id).

L'attribut **[EnableQuery]** permet aux clients de modifier la requête en utilisant des options de requête telles que $ filter, $ sort et $ page. Pour plus d'informations, voir Prise en charge des options de requête OData .

Ajout d'une entité à l'ensemble d'entités

Pour permettre aux clients d'ajouter un nouveau produit à la base de données, ajoutez la méthode suivante à **ProductsController** .

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

Mise à jour d'une entité

OData prend en charge deux sémantiques différentes pour mettre à jour une entité, PATCH et PUT.

- PATCH effectue une mise à jour partielle. Le client spécifie uniquement les propriétés à mettre à jour.
- PUT remplace l'entité entière.

L'inconvénient de PUT est que le client doit envoyer des valeurs pour toutes les propriétés de l'entité, y compris celles qui ne changent pas. La spécification OData indique que PATCH est préféré.

En tout cas, voici le code pour les méthodes PATCH et 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);
}
```

Dans le cas de PATCH, le contrôleur utilise le type **Delta <T>** pour suivre les modifications.

Supprimer une entité

Pour permettre aux clients de supprimer un produit de la base de données, ajoutez la méthode suivante à **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);
}
```

Lire OData avec l'API Web Asp.net en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/6019/odata-avec-l-api-web-asp-net

Chapitre 10: Routage d'attribut dans WebAPI

Introduction

Comme son nom l'indique, cela utilise des attributs pour router. Cela donne à l'utilisateur plus de contrôle sur les URI dans WebAPI. Par exemple, vous pouvez décrire des hiérarchies de la ressource. Toutefois, le «routage conventionnel» antérieur est entièrement pris en charge. Les utilisateurs peuvent aussi avoir un mélange des deux.

Syntaxe

- [RoutePrefix ("api / books")] - pour la classe du contrôleur
- [Route ("getById")]] - pour les actions
- [Route ("~ / api / authors / {authorId: int} / books")] - pour remplacer le préfixe de route

Paramètres

Le nom du paramètre	Détails
RoutePrefix	attribuer à la classe de contrôleur. tous les préfixes d'url courants dans les actions sont mis en boîte ici. prend la chaîne en entrée
Route	attribuer aux actions du contrôleur. chaque action aura son itinéraire associé (pas nécessairement)
Route ("~ / api /")	cela remplace le préfixe de route

Remarques

Actuellement, les chemins d'attributs ne disposent pas de `Controller specific Message Handlers`. Comme il n'y a aucun moyen de spécifier quel gestionnaire exécuter pour quelle route au moment de la déclaration. Ceci est possible dans le `Conventional Routing`.

Examples

Routage d'attribut de base

Ajoutez simplement un attribut à l'action du contrôleur

```
[Route("product/{productId}/customer")]
public IQueryable<Product> GetProductsByCustomer(int productId)
{
```

```
    //action code goes here
}
```

Cette `productId=1` sera interrogée sous la forme `/product/1/customer` et `productId=1` sera envoyé à l'action du contrôleur.

Assurez-vous que celui dans '{}' et le paramètre d'action sont identiques. `productId` dans ce cas.

Avant d'utiliser ceci, vous devez spécifier que vous utilisez le routage d'attribut par:

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

Attribut de préfixe de route

Dans les cas où vous avez besoin d'une partie commune de l'itinéraire pour toutes les routes dans un contrôleur, l'attribut `RoutePrefix` est utilisé.

Dans l'exemple ci-dessous, une partie api / students du code est commune et nous pouvons donc définir `RoutePrefix` et éviter de l'utiliser à plusieurs reprises.

```
[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
}

}
```

Lire Routage d'attribut dans WebAPI en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/9440/routage-d-attribut-dans-webapi

Chapitre 11: Routage URL Web API

Examples

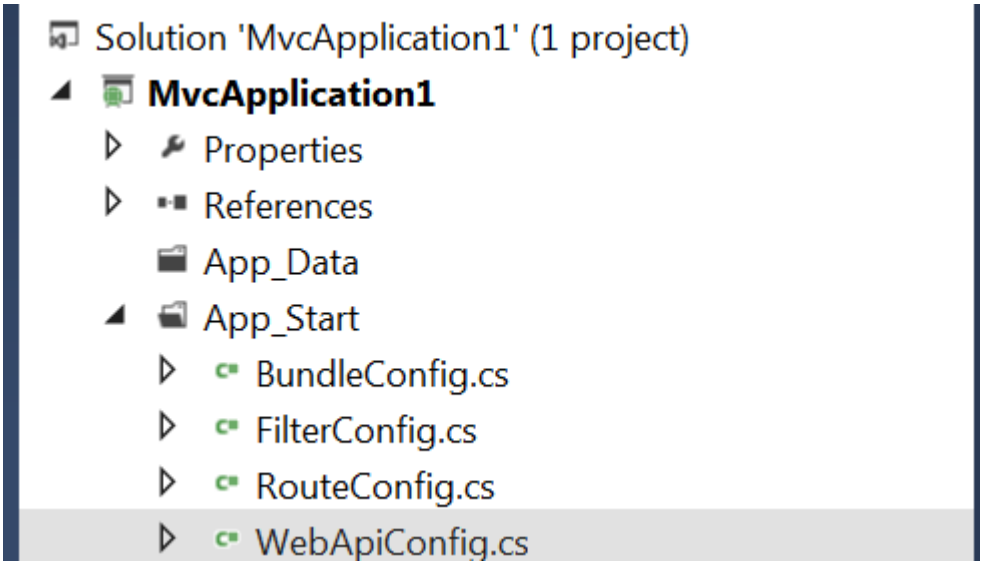
Comment fonctionne le routage dans asp.net webapi

Dans l'API Web ASP.NET, un contrôleur est une classe qui gère les requêtes HTTP. Les méthodes publiques du contrôleur sont appelées méthodes d'action ou simplement actions.

Lorsque l'infrastructure de l'API Web reçoit une demande, elle achemine la demande vers une action. Pour déterminer quelle action appeler, le framework utilise une table de routage. Le modèle de projet Visual Studio pour Web API crée une route par défaut:

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

Cette route est définie dans le fichier WebApiConfig.cs, qui est placé dans le répertoire App_Start:



Chaque entrée de la table de routage contient un modèle de route. Le modèle de route par défaut pour Web API est " **api / {controller} / {id}** ". Dans ce modèle, " **api** " est un segment de chemin littéral, et { **controller** } et { **id** } sont des variables d'espace réservé.

Lorsque l'infrastructure d'API Web reçoit une requête HTTP, elle tente de faire correspondre l'URI à l'un des modèles de route de la table de routage. Si aucune route ne correspond, le client reçoit une erreur 404.

Par exemple, les URI suivants correspondent à l'itinéraire par défaut:

- / api / valeurs
- / api / valeurs / 1

Cependant, l'URI suivant ne correspond pas, car il manque le segment " **api** ":

- / valeurs / 1

Une fois qu'un itinéraire correspondant est trouvé, Web API sélectionne le contrôleur et l'action:

- Pour trouver le contrôleur, Web API ajoute "Controller" à la valeur de la variable {controller}.
- Pour rechercher l'action, l'API Web examine la méthode HTTP, puis recherche une action dont le nom commence par ce nom de méthode HTTP. Par exemple, avec une requête GET, Web API recherche une action qui commence par "Get ...", par exemple "GetEmployee" ou "GetAllEmployees". Cette convention s'applique uniquement aux méthodes GET, POST, PUT et DELETE.

Vous pouvez activer d'autres méthodes HTTP en utilisant des attributs sur votre contrôleur. Nous en verrons un exemple plus tard.

- Les autres variables d'espace réservé dans le modèle de route, telles que {id}, sont mappées aux paramètres d'action.

Méthodes HTTP Au lieu d'utiliser la convention d'attribution de nom pour les méthodes HTTP, vous pouvez spécifier explicitement la méthode HTTP pour une action en décorant la méthode d'action avec l'attribut HttpGet, HttpPut, HttpPost ou HttpDelete.

Dans l'exemple suivant, la méthode EmployeeGetEmployee est mappée aux requêtes GET:

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

Pour autoriser plusieurs méthodes HTTP pour une action ou pour autoriser des méthodes HTTP autres que GET, PUT, POST et DELETE, utilisez l'attribut AcceptVerbs, qui prend une liste de méthodes HTTP.

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

Nom du routage par action

Avec le modèle de routage par défaut, Web API utilise la méthode HTTP pour sélectionner l'action. Cependant, vous pouvez également créer un itinéraire où le nom de l'action est inclus dans l'URI:

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

```
);
```

Dans ce modèle de route, le paramètre {action} nomme la méthode d'action sur le contrôleur. Avec ce style de routage, utilisez des attributs pour spécifier les méthodes HTTP autorisées. Par exemple, supposons que votre contrôleur utilise la méthode suivante:

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

Dans ce cas, une demande GET pour « **api / Employees / GetAllEmployees** » correspondrait à la méthode GetAllEmployees.

Vous pouvez remplacer le nom de l'action en utilisant l'attribut ActionName. Dans l'exemple suivant, deux actions correspondent à " **api / Employees / ShowAllEmployees / id** . L'une prend en charge GET et l'autre prend en charge POST:

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

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

}
```

Non-actions

Nous pouvons empêcher une méthode d'être appelée en tant qu'action en utilisant l'attribut NonAction. Cela indique au framework que la méthode n'est pas une action, même si elle correspondrait aux règles de routage.

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

Exemples de routage basés sur des verbes.

La même URL pour différentes méthodes http agit différemment. Ci-dessous, un tableau représentant le même.

VERBE HTTP	URL	LA DESCRIPTION
OBTENIR	/ api / étudiants	Retourne tous les étudiants
OBTENIR	/ api / étudiants / 5	Retourne les détails de l'id de l'étudiant = 5

VERBE HTTP	URL	LA DESCRIPTION
POSTER	/ api / étudiants	Ajouter un nouvel étudiant
METTRE	/ api / étudiants / 5	Mettre à jour l'élève avec Id = 5
EFFACER	/ api / étudiants / 5	Supprimer l'élève avec Id = 5

Lire Routage URL Web API en ligne: https://riptutorial.com/fr/asp-net-web-api/topic/2432/routage-url-web-api

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec asp.net-web-api	Arif, BehrouzMoslem, Community, The_Outsider
2	ASP.NET WEB API Activer CORS	Alexei, Oluwafemi
3	ASP.NET Web API MediaTypeFormatter	Keyur Ramoliya
4	Configurez une application Web API pour répondre avec des données JSON jolies / formatées par défaut	Patrick
5	Création d'un ActionFilterAttribute personnalisé	Antarr Byrd, Blanthor
6	Démarrage rapide: Travailler avec JSON	Brett Veenstra
7	Mise en cache	Sibeesh Venu
8	Négociation du contenu de l'API Web ASP.NET	Amit Shahani, Keyur Ramoliya
9	OData avec l'API Web Asp.net	Yushell
10	Routage d'attribut dans WebAPI	Ajay Aradhya, The_Outsider
11	Routage URL Web API	ravindra, riteshmeher, The_Outsider