عنوان: **بررسی مقدمات کتابخانه‌ی JSON.NET**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۳ ۱۹:۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, JSON, ASP.NET Web API, json.net

چرا JSON.NET؟

JSON.NET یک کتابخانه‌ی سورس باز کار با اشیاء JSON در دات نت است. تاریخچه‌ی آن به 8 سال قبل بر می‌گردد و توسط یک برنامه نویس نیوزیلندی به نام James Newton King تهیه شده‌است. اولین نگارش آن در سال 2006 ارائه شد؛ مقارن با زمانی که اولین استاندارد JSON نیز ارائه گردید.

این کتابخانه از آن زمان تا کنون، 6 میلیون بار دانلود شده‌است و به علت کیفیت بالای آن، این روزها پایه اصلی بسیاری از کتابخانه‌ها و فریم ورک‌های دات نتی می‌باشد؛ مانند RavenDB تا ASP.NET Web API و SignalR مایکروسافت و همچنین گوگل نیز از آن جهت تدارک کلاینت‌های کار با API خود استفاده می‌کنند.

هرچند دات نت برای نمونه در نگارش سوم آن جهت مصارف WCF کلاسی را به نام DataContractJsonSerializer ارائه کرد، اما کار کردن با آن محدود است به فرمت خاص WCF به همراه عدم انعطاف پذیری و سادگی کار با آن. به علاوه باید درنظر داشت که JSON.NET از دات نت 2 به بعد تا مونو، Win8 و ویندوز فون را نیز پشتیبانی می‌کند.

برای نصب آن نیز کافی است دستور ذیل را در کنسول پاورشل نیوگت اجرا کنید:

```
PM> install-package Newtonsoft.Json
```

معماری JSON.NET

کتابخانه‌ی JSON.NET از سه قسمت عمده تشکیل شده‌است:

الف) JsonSerializer

ب) LINQ to JSON

ج) JSON Schema

الف) JsonSerializer

کار JsonSerializer تبدیل اشیاء دات نتی به JSON و برعکس است. مزیت مهم آن امکانات قابل توجه تنظیم عملکرد و خروجی آن می‌باشد که این تنظیمات را به شکل ویژگی‌های خواص نیز می‌توان اعمال نمود. به علاوه امکان سفارشی سازی هر کدام نیز توسط کلاسی به نام JsonConverter، پیش بینی شده‌است.

یک مثال:

```
var roles = new List<string>
{
   "Admin",
   "User"
};
string json = JsonConvert.SerializeObject(roles, Formatting.Indented);
```

در اینجا نحوه‌ی استفاده از JSON.NET را جهت تبدیل یک شیء دات نتی، به معادل JSON آن مشاهده می‌کنید. اعمال تنظیم Formatting.Indented سبب خواهد شد تا خروجی آن دارای Indentation باشد. برای نمونه اگر در برنامه‌ی خود قصد دارید فرمت JSON تو در توی را به نحو زیبا و خوانایی نمایش دهید یا چاپ کنید، همین تنظیم ساده کافی خواهد بود.

و یا در مثال ذیل استفاده از یک anonymous object را مشاهده می‌کنید:

```
var jsonString = JsonConvert.SerializeObject(new
{
   Id =1,
   Name = "Test"
}, Formatting.Indented);
```

به صورت پیش فرض تنها خواص عمومی کلاس‌ها توسط JSON.NET تبدیل خواهند شد.

تنظیمات پیشرفته‌تر JSON.NET

مزیت مهم JSON.NET بر سایر کتابخانه‌های موجود مشابه، قابلیت‌های سفارشی سازی قابل توجه آن است. در مثال ذیل نحوه‌ی معرفی JsonSerializerSettings را مشاهده می‌نمائید:

```
var jsonData = JsonConvert.SerializeObject(new
{
   Id = 1,
   Name = "Test",
   DateTime = DateTime.Now
}, new JsonSerializerSettings
{
   Formatting = Formatting.Indented,
   Converters =
   {
      new JavaScriptDateTimeConverter()
   }
});
```

در اینجا با استفاده از تنظیم JavaScriptDateTimeConverter، می‌توان خروجی DateTime استانداردی را به مصرف کنندگان جاوا اسکریپتی سمت کاربر ارائه داد؛ با خروجی ذیل:

```
{
  "Id": 1,
  "Name": "Test",
  "DateTime": new Date(1409821985245)
}
```

نوشتن خروجی JSON در یک استریم

خروجی متد JsonConvert.SerializeObject یک رشته‌است که در صورت نیاز به سادگی توسط متد File.WriteAllText در یک فایل قابل ذخیره می‌باشد. اما برای رسیدن به حداکثر کارآیی و سرعت می‌توان از استریم‌ها نیز استفاده کرد:

```
using (var stream = File.CreateText(@"c:\output.json"))
{
    var jsonSerializer = new JsonSerializer
   {
      Formatting = Formatting.Indented
   };
   jsonSerializer.Serialize(stream, new
   {
     Id = 1,
     Name = "Test",
     DateTime = DateTime.Now
   });
}
```

کلاس JsonSerializer و متد Serialize آن یک استریم را نیز جهت نوشتن خروجی می‌پذیرند. برای مثال response.Output برنامه‌های وب نیز یک استریم است و در اینجا نوشتن مستقیم در استریم بسیار سریعتر است از تبدیل شیء به رشته و سپس ارائه خروجی آن؛ زیرا سربار تهیه رشته JSON از آن حذف می‌گردد و نهایتا GC کار کمتری را باید انجام دهد.

تبدیل JSON رشته‌ای به اشیاء دات نت

اگر رشته‌ی jsonData ایی را که پیشتر تولید کردیم، بخواهیم تبدیل به نمونه‌ای از شیء User ذیل کنیم:

```
public class User
{
```

```
   public int Id { set; get; }
   public string Name { set; get; }
   public DateTime DateTime { set; get; }
}
```

خواهیم داشت:

```
var user = JsonConvert.DeserializeObject<User>(jsonData);
```

در اینجا از متد DeserializeObject به همراه مشخص سازی صریح نوع شیء نهایی استفاده شده‌است.
البته در اینجا با توجه به استفاده از JavaScriptDateTimeConverter برای تولید jsonData، نیاز است چنین تنظیمی را نیز در حالت DeserializeObject مشخص کنیم:

```
var user = JsonConvert.DeserializeObject<User>(jsonData, new JsonSerializerSettings
{
   Converters = {  new JavaScriptDateTimeConverter() }
});
```

مقدار دهی یک نمونه یا وهله‌ی از پیش موجود

متد JsonConvert.DeserializeObject یک شیء جدید را ایجاد می‌کند. اگر قصد دارید صرفا تعدادی از خواص یک وهله‌ی موجود، توسط JSON.NET مقدار دهی شوند از متد PopulateObject استفاده کنید:

```
JsonConvert.PopulateObject(jsonData, user);
```

کاهش حجم JSON تولیدی

زمانیکه از متد JsonConvert.SerializeObject استفاده می‌کنیم، تمام خواص عمومی تبدیل به معادل JSON آن‌ها خواهند شد؛ حتی خواصی که مقدار ندارند. این خواص در خروجی JSON، با مقدار null مشخص می‌شوند. برای حذف این خواص از خروجی JSON نهایی تنها کافی است در تنظیمات JsonSerializerSettings، مقدار NullValueHandling = NullValueHandling.Ignore مشخص گردد.

```
var jsonData = JsonConvert.SerializeObject(object, new JsonSerializerSettings
{
   NullValueHandling = NullValueHandling.Ignore,
   Formatting = Formatting.Indented
});
```

مورد دیگری که سبب کاهش حجم خروجی نهایی خواهد شد، تنظیم DefaultValueHandling = DefaultValueHandling.Ignore است. در این حالت کلیه خواصی که دارای مقدار پیش فرض خودشان هستند، در خروجی JSON ظاهر نخواهند شد. مثلا مقدار پیش فرض خاصیت int مساوی صفر است. در این حالت کلیه خواص از نوع int که دارای مقدار صفر می‌باشند، در خروجی قرار نمی‌گیرند.
به علاوه حذف Formatting = Formatting.Indented نیز توصیه می‌گردد. در این حالت فشرده‌ترین خروجی ممکن حاصل خواهد شد.

مدیریت ارث بری توسط JSON.NET

در مثال ذیل کلاس کارمند و کلاس مدیر را که خود نیز در اصل یک کارمند می‌باشد، ملاحظه می‌کنید:

```
public class Employee
{
    public string Name { set; get; }
}

public class Manager : Employee
{
    public IList<Employee> Reports { set; get; }
}
```

در اینجا هر مدیر لیست کارمندانی را که به او گزارش می‌دهند نیز به همراه دارد. در ادامه نمونه‌ای از مقدار دهی این اشیاء ذکر شده‌اند:

```
var employee = new Employee { Name = "User1" };
var manager1 = new Manager { Name = "User2" };
var manager2 = new Manager { Name = "User3" };
manager1.Reports = new[] { employee, manager2 };
manager2.Reports = new[] { employee };
```

با فراخوانی

```
var list = JsonConvert.SerializeObject(manager1, Formatting.Indented);
```

یک چنین خروجی JSON ایی حاصل می‌شود:

```
{
  "Reports": [
    {
      "Name": "User1"
    },
    {
      "Reports": [
        {
          "Name": "User1"
        }
      ],
      "Name": "User3"
    }
  ],
  "Name": "User2"
}
```

این خروجی JSON جهت تبدیل به نمونه‌ی معادل دات نتی خود، برای مثال جهت رسیدن به manager1 در کدهای فوق، چندین مشکل را به همراه دارد:

- در اینجا مشخص نیست که این اشیاء، کارمند هستند یا مدیر. برای مثال مشخص نیست User2 چه نوعی دارد و باید به کدام شیء نگاشت شود.
- مشکل دوم در مورد کاربر User1 است که در دو قسمت تکرار شده‌است. این شیء JSON اگر به نمونه‌ی معادل دات نتی خود نگاشت شود، به دو وهله از User1 خواهیم رسید و نه یک وهله‌ی اصلی که سبب تولید این خروجی JSON شده‌است.

برای حل این دو مشکل، تغییرات ذیل را می‌توان به JSON.NET اعمال کرد:

```
var list = JsonConvert.SerializeObject(manager1, new JsonSerializerSettings
{
   Formatting = Formatting.Indented,
   TypeNameHandling = TypeNameHandling.Objects,
   PreserveReferencesHandling = PreserveReferencesHandling.Objects
});
```

با این خروجی:

```
{
  "$id": "1",
  "$type": "JsonNetTests.Manager, JsonNetTests",
  "Reports": [
    {
      "$id": "2",
      "$type": "JsonNetTests.Employee, JsonNetTests",
      "Name": "User1"
    },
    {
      "$id": "3",
      "$type": "JsonNetTests.Manager, JsonNetTests",
      "Reports": [
        {
          "$ref": "2"
        }
      ],
      "Name": "User3"
    }
  ],
  "Name": "User2"
}
```

- با تنظیم TypeNameHandling = TypeNameHandling.Objects سبب خواهیم شد تا خاصیت اضافه‌ای به نام $type به خروجی JSON اضافه شود. این نوع، در حین فراخوانی متد JsonConvert.DeserializeObject جهت تشخیص صحیح نگاشت اشیاء بکار گرفته خواهد شد و اینبار مشخص است که کدام شیء، کارمند است و کدامیک مدیر.
- با تنظیم PreserveReferencesHandling = PreserveReferencesHandling.Objects شماره Id خودکاری نیز به خروجی JSON اضافه می‌گردد. اینبار اگر به گزارش دهنده‌ها با دقت نگاه کنیم، مقدار $ref=2 را خواهیم دید. این مورد سبب می‌شود تا در حین نگاشت نهایی، دو وهله متفاوت از شیء با Id=2 تولید نشود.

باید دقت داشت که در حین استفاده از JsonConvert.DeserializeObject نیز باید JsonSerializerSettings یاد شده، تنظیم شوند.

ویژگی‌های قابل تنظیم در JSON.NET

علاوه بر JsonSerializerSettings که از آن صحبت شد، در JSON.NET امکان تنظیم یک سری از ویژگی‌ها به ازای خواص مختلف نیز وجود دارند.
- برای نمونه ویژگی JsonIgnore معروفترین آن‌ها است:

```
public class User
{
   public int Id { set; get; }

   [JsonIgnore]
   public string Name { set; get; }

   public DateTime DateTime { set; get; }
}
```

JsonIgnore سبب می‌شود تا خاصیتی در خروجی نهایی JSON تولیدی حضور نداشته باشد و از آن صرفنظر شود.

- با استفاده از ویژگی JsonProperty اغلب مواردی را که پیشتر بحث کردیم مانند TypeNameHandling، NullValueHandling و غیره، می‌توان تنظیم نمود. همچنین گاهی از اوقات کتابخانه‌های جاوا اسکریپتی سمت کاربر، از اسامی خاصی که از روش‌های نامگذاری دات نتی پیروی نمی‌کنند، در طراحی خود استفاده می‌کنند. در اینجا می‌توان نام خاصیت نهایی را که قرار است رندر شود نیز صریحا مشخص کرد. برای مثال:

```
[JsonProperty(PropertyName = "m_name", NullValueHandling = NullValueHandling.Ignore)]
public string Name { set; get; }
```

همچنین در اینجا امکان تنظیم Order نیز وجود دارد. برای مثال مشخص کنیم که خاصیت X در ابتدا قرار گیرد و پس از آن خاصیت Y رندر شود.

- استفاده از ویژگی JsonObject به همراه مقدار OptIn آن به این معنا است که از کلیه خواصی که دارای ویژگی JsonProperty نیستند، صرفنظر شود. حالت پیش فرض آن OptOut است؛ یعنی تمام خواص عمومی در خروجی JSON حضور خواهند داشت منهای مواردی که با JsonIgnore مزین شوند.

```
[JsonObject(MemberSerialization.OptIn)]
public class User
{
    public int Id { set; get; }

    [JsonProperty]
    public string Name { set; get; }

    public DateTime DateTime { set; get; }
}
```

- با استفاده از ویژگی JsonConverter می‌توان نحوه‌ی رندر شدن مقدار خاصیت را سفارشی سازی کرد. برای مثال:

```
[JsonConverter(typeof(JavaScriptDateTimeConverter))]
public DateTime DateTime { set; get; }
```

تهیه یک JsonConverter سفارشی

با استفاده از JsonConverterها می‌توان کنترل کاملی را بر روی اعمال serialization و deserialization مقادیر خواص اعمال کرد. مثال زیر را در نظر بگیرید:

```
public class HtmlColor
{
   public int Red { set; get; }
   public int Green { set; get; }
   public int Blue { set; get; }
}

var colorJson = JsonConvert.SerializeObject(new HtmlColor
{
  Red = 255,
  Green = 0,
  Blue = 0
}, Formatting.Indented);
```

در اینجا علاقمندیم، در حین عملیات serialization، بجای اینکه مقادیر اجزای رنگ تهیه شده به صورت int نمایش داده شوند، کل رنگ با فرمت hex رندر شوند. برای اینکار نیاز است یک JsonConverter سفارشی را تدارک دید:

```
public class HtmlColorConverter : JsonConverter
    {

        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(HtmlColor);
        }

        public override object ReadJson(JsonReader reader, Type objectType,
                                        object existingValue, JsonSerializer serializer)
        {
            throw new NotSupportedException();
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
        {
            var color = value as HtmlColor;
            if (color == null)
```

```
                return;

            writer.WriteValue("#" + color.Red.ToString("X2")
                + color.Green.ToString("X2") + color.Blue.ToString("X2"));
        }
    }
```

کار با ارث بری از کلاس پایه JsonConverter شروع می‌شود. سپس باید تعدادی از متدهای این کلاس پایه را بازنویسی کرد. در متد CanConvert اعلام می‌کنیم که تنها اشیایی از نوع کلاس HtmlColor را قرار است پردازش کنیم. سپس در متد WriteJson منطق سفارشی خود را می‌توان پیاده سازی کرد.
از آنجائیکه این تبدیلگر صرفا قرار است برای حالت serialization استفاده شود، قسمت ReadJson آن پیاده سازی نشده‌است.

در آخر برای استفاده از آن خواهیم داشت:

```
var colorJson = JsonConvert.SerializeObject(new HtmlColor
{
  Red = 255,
  Green = 0,
  Blue = 0
},  new JsonSerializerSettings
    {
      Formatting = Formatting.Indented,
      Converters = { new HtmlColorConverter() }
    });
```

نظرات خوانندگان

نویسنده: افتابی
تاریخ: ۱۳۹۳/۰۶/۱۳ ۲۱:۵۴

سلام؛ من مطالب مربوطه رو خوندم فقط اینکه توی یه صفحه rozar در mvc من به چه نحو میتونم از آن استفاده کنم ، حتی توی سایت خودش هم رفتم و sample ها رو دیدم فقط میخوام در یک پروژه به چه نحو ازش استفاده کنم و کجا کارش ببرم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۳ ۲۱:۵۹

در یک اکشن متد، بجای return Json پیش فرض و توکار، می‌شود نوشت:

```
return Content(JsonConvert.SerializeObject(obj));
```

البته این ساده‌ترین روش استفاده از آن است؛ برای مقاصد Ajax ایی.
و یا برای ذکر Content type می‌توان به صورت زیر عمل کرد:

```
return new ContentResult
{
  Content = JsonConvert.SerializeObject(obj),
  ContentType = "application/json"
};
```

نویسنده: رحمت اله رضایی
تاریخ: ۱۳۹۳/۰۶/۱۴ ۱۴:۳

"ASP.NET Web API و SignalR از این کتابخانه استفاده می‌کنند". دلیلی دارد هنوز ASP.NET MVC از این کتابخانه استفاده نکرده است؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۴ ۱۴:۱۹

- تا ASP.NET MVC 5 از JavaScriptSerializer در JsonResult استفاده می‌شود.
- در نگارش بعدی ASP.NET MVC که با Web API یکی شده (یعنی در یک کنترلر هم می‌توانید ActionResult داشته باشید و هم خروجی‌های متداول Web API را با هم) اینبار تامین کننده‌ی JsonResult از طریق تزریق وابستگی‌ها تامین می‌شود و می‌تواند هر کتابخانه‌ای که صلاح می‌دانید باشد. البته یک مقدار پیش فرض هم دارد که دقیقا از JSON.NET استفاده می‌کند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۷/۱۸ ۱۲:۳۵

یک نکته‌ی تکمیلی

استفاده از استریم‌ها برای کار با فایل‌ها در JSON.NET

```
public static T DeserializeFromFile<T>(string filePath, JsonSerializerSettings settings = null)
        {
            if (!File.Exists(filePath))
                return default(T);

            using (var fileStream = File.OpenRead(filePath))
            {
                using (var streamReader = new StreamReader(fileStream))
                {
                    using (var reader = new JsonTextReader(streamReader))
                    {
                        var serializer = settings == null ? JsonSerializer.Create() :
JsonSerializer.Create(settings);
```

```
                        return serializer.Deserialize<T>(reader);
                    }
                }
            }
        }

        public static void SerializeToFile(string filePath, object data, JsonSerializerSettings
settings = null)
        {
            using (var fileStream = new FileStream(filePath, FileMode.Create))
            {
                using (var streamReader = new StreamWriter(fileStream))
                {
                    using (var reader = new JsonTextWriter(streamReader))
                    {
                        var serializer = settings == null ? JsonSerializer.Create() :
JsonSerializer.Create(settings);
                        serializer.Serialize(reader, data);
                    }
                }
            }
        }
```

عنوان: **تنظیمات و نکات کاربردی کتابخانه‌ی JSON.NET**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۴ ۱۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, JSON, ASP.NET Web API, json.net, Entity framework

پس از بررسی مقدماتی امکانات کتابخانه‌ی JSON.NET، در ادامه به تعدادی از تنظیمات کاربردی آن با ذکر مثال‌هایی خواهیم پرداخت.

گرفتن خروجی CamelCase از JSON.NET

یک سری از کتابخانه‌های جاوا اسکریپتی سمت کلاینت، به نام‌های خواص CamelCase نیاز دارند و حالت پیش فرض اصول نامگذاری خواص در دات نت عکس آن است. برای مثال بجای UserName به userName نیاز دارند تا بتوانند صحیح کار کنند.
روش اول حل این مشکل، استفاده از ویژگی JsonProperty بر روی تک تک خواص و مشخص کردن نام‌های مورد نیاز کتابخانه‌ی جاوا اسکریپتی به صورت صریح است.
روش دوم، استفاده از تنظیمات ContractResolver می‌باشد که با تنظیم آن به CamelCasePropertyNamesContractResolver به صورت خودکار به تمامی خواص به صورت یکسانی اعمال می‌گردد:

```
var json = JsonConvert.SerializeObject(obj, new JsonSerializerSettings
{
   ContractResolver = new CamelCasePropertyNamesContractResolver()
});
```

درج نام‌های المان‌های یک Enum در خروجی JSON

اگر یکی از عناصر در حال تبدیل به JSON، از نوع enum باشد، به صورت پیش فرض مقدار عددی آن در JSON نهایی درج می‌گردد:

```
using Newtonsoft.Json;

namespace JsonNetTests
{
    public enum Color
    {
        Red,
        Green,
        Blue,
        White
    }

    public class Item
    {
        public string Name { set; get; }
        public Color Color { set; get; }
    }

    public class EnumTests
    {
        public string GetJson()
        {
            var item = new Item
            {
                Name = "Item 1",
                Color = Color.Blue
            };

            return JsonConvert.SerializeObject(item, Formatting.Indented);
        }
    }
}
```

با این خروجی:

```
{
  "Name": "Item 1",
  "Color": 2
}
```

اگر علاقمند هستید که بجای عدد 2، دقیقا مقدار Blue در خروجی JSON درج گردد، می‌توان به یکی از دو روش ذیل عمل کرد:
الف) مزین کردن خاصیت از نوع enum به ویژگی JsonConverter از نوع StringEnumConverter:

```
[JsonConverter(typeof(StringEnumConverter))]
  public Color Color { set; get; }
```

ب) و یا اگر می‌خواهید این تنظیم به تمام خواص از نوع enum به صورت یکسانی اعمال شود، می‌توان نوشت:

```
return JsonConvert.SerializeObject(item, new JsonSerializerSettings
{
   Formatting = Formatting.Indented,
   Converters = { new StringEnumConverter() }
});
```

تهیه خروجی JSON از مدل‌های مرتبط، بدون Stack overflow

دو کلاس گروه‌های محصولات و محصولات ذیل را درنظر بگیرید:

```
public class Category
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public virtual ICollection<Product> Products { get; set; }

        public Category()
        {
            Products = new List<Product>();
        }
    }

    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public virtual Category Category { get; set; }
    }
```

این نوع طراحی در Entity framework بسیار مرسوم است. در اینجا طرف‌های دیگر یک رابطه، توسط خاصیتی virtual معرفی می‌شوند که به آن‌ها خواص راهبری یا navigation properties هم می‌گویند.
با توجه به این دو کلاس، سعی کنید مثال ذیل را اجرا کرده و از آن، خروجی JSON تهیه کنید:

```
using System.Collections.Generic;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;

namespace JsonNetTests
{
    public class SelfReferencingLoops
    {
        public string GetJson()
        {
            var category = new Category
            {
                Id = 1,
                Name = "Category 1"
            };
            var product = new Product
```

```
            {
                Id = 1,
                Name = "Product 1"
            };

            category.Products.Add(product);
            product.Category = category;

            return JsonConvert.SerializeObject(category, new JsonSerializerSettings
            {
                Formatting = Formatting.Indented,
                Converters = { new StringEnumConverter() }
            });
        }
    }
}
```

برنامه با این استثناء متوقف می‌شود:

```
An unhandled exception of type 'Newtonsoft.Json.JsonSerializationException' occurred in
Newtonsoft.Json.dll
Additional information: Self referencing loop detected for property 'Category' with type
'JsonNetTests.Category'. Path 'Products[0]'.
```

اصل خطای معروف فوق «Self referencing loop detected» است. در اینجا کلاس‌هایی که به یکدیگر ارجاع می‌دهند، در حین عملیات Serialization سبب بروز یک حلقه‌ی بازگشتی بی‌نهایت شده و در آخر، برنامه با خطای stack overflow خاتمه می‌یابد.

راه حل اول:

به تنظیمات JSON.NET، مقدار ReferenceLoopHandling = ReferenceLoopHandling.Ignore را اضافه کنید تا از حلقه‌ی بازگشتی بی‌پایان جلوگیری شود:

```
return JsonConvert.SerializeObject(category, new JsonSerializerSettings
{
   Formatting = Formatting.Indented,
   ReferenceLoopHandling = ReferenceLoopHandling.Ignore,
   Converters = { new StringEnumConverter() }
});
```

راه حل دوم:

به تنظیمات JSON.NET، مقدار PreserveReferencesHandling = PreserveReferencesHandling.Objects را اضافه کنید تا مدیریت ارجاعات اشیاء توسط خود JSON.NET انجام شود:

```
return JsonConvert.SerializeObject(category, new JsonSerializerSettings
{
   Formatting = Formatting.Indented,
   PreserveReferencesHandling = PreserveReferencesHandling.Objects,
   Converters = { new StringEnumConverter() }
});
```

خروجی حالت دوم به این شکل است:

```
{
  "$id": "1",
  "Id": 1,
  "Name": "Category 1",
  "Products": [
    {
      "$id": "2",
      "Id": 1,
      "Name": "Product 1",
      "Category": {
        "$ref": "1"
      }
    }
```

```
  ]
}
```

همانطور که ملاحظه می‌کنید، دو خاصیت $id و $ref توسط JSON.NET به خروجی JSON اضافه شده‌است تا توسط آن بتواند ارجاعات و نمونه‌های اشیاء را تشخیص دهد.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۳ ۱۵:۱۷

گرفتن خروجی مرتب شده بر اساس نام خواص (جهت مقاصد نمایشی):

تعریف DefaultContractResolver :

```
public class OrderedContractResolver : DefaultContractResolver
    {
        protected override IList<JsonProperty> CreateProperties(
            System.Type type, MemberSerialization memberSerialization)
        {
            return base.CreateProperties(type, memberSerialization).OrderBy(p =>
p.PropertyName).ToList();
        }
    }
```

و بعد معرفی آن به نحو ذیل:

```
return JsonConvert.SerializeObject(data, new JsonSerializerSettings
            {
                ContractResolver = new OrderedContractResolver()
            });
```

عنوان: **استفاده از JSON.NET در ASP.NET MVC**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۵ ۲۳:۲۵
آدرس: www.dotnettips.info
گروه‌ها: MVC, JSON, json.net

تا نگارش فعلی ASP.NET MVC، یعنی نگارش 5 آن، به صورت توکار از JavaScriptSerializer برای پردازش JSON کمک گرفته می‌شود. این کلاس نسبت به JSON.NET هم کندتر است و هم قابلیت سفارشی سازی آنچنانی ندارد. برای مثال مشکل Self referencing loop را نمی‌تواند مدیریت کند.
برای استفاده از JSON.NET در یک اکشن متد، به صورت معمولی می‌توان به نحو ذیل عمل کرد:

```
[HttpGet]
        public ActionResult GetSimpleJsonData()
        {
            return new ContentResult
            {
                Content = JsonConvert.SerializeObject(new { id = 1 }),
                ContentType = "application/json",
                ContentEncoding = Encoding.UTF8
            };
        }
```

در اینجا با استفاده از متد JsonConvert.SerializeObject، اطلاعات شیء مدنظر تبدیل به یک رشته شده و سپس با content type مناسبی در اختیار مصرف کننده قرار می‌گیرد.
اگر بخواهیم این عملیات را کمی بهینه‌تر کنیم، نیاز است بتوانیم از استریم‌ها استفاده کرده و خروجی JSON را بدون تبدیل به رشته، مستقیما در استریم response.Output بنویسیم. با اینکار به سرعت بیشتر و همچنین مصرف منابع کمتری خواهیم رسید.
نمونه‌ای از این پیاده سازی را در ذیل مشاهده می‌کنید:

```
using System;
using System.Web.Mvc;
using Newtonsoft.Json;

namespace MvcJsonNetTests.Utils
{
    public class JsonNetResult : JsonResult
    {
        public JsonNetResult()
        {
            Settings = new JsonSerializerSettings { ReferenceLoopHandling = ReferenceLoopHandling.Error
};
        }

        public JsonSerializerSettings Settings { get; set; }

        public override void ExecuteResult(ControllerContext context)
        {
            if (context == null)
                throw new ArgumentNullException("context");

            if (this.JsonRequestBehavior == JsonRequestBehavior.DenyGet &&
                string.Equals(context.HttpContext.Request.HttpMethod, "GET",
StringComparison.OrdinalIgnoreCase))
            {
                throw new InvalidOperationException("To allow GET requests, set JsonRequestBehavior to
AllowGet.");
            }

            if (this.Data == null)
                return;

            var response = context.HttpContext.Response;
            response.ContentType = string.IsNullOrEmpty(this.ContentType) ? "application/json" :
this.ContentType;

            if (this.ContentEncoding != null)
                response.ContentEncoding = this.ContentEncoding;

            var serializer = JsonSerializer.Create(this.Settings);
            using (var writer = new JsonTextWriter(response.Output))
            {
```

```
                serializer.Serialize(writer, Data);
                writer.Flush();
            }
        }
    }
}
```

اگر دقت کنید، کار با ارث بری از JsonResult توکار ASP.NET MVC شروع شده‌است. کدهای ابتدای متد ExecuteResult با کدهای اصلی JsonResult یکی هستند. فقط انتهای کار بجای استفاده از JavaScriptSerializer، از JSON.NET استفاده شده‌است.
در این حالت برای استفاده از این Action Result جدید می‌توان نوشت:

```
[HttpGet]
        public ActionResult GetJsonData()
        {
            return new JsonNetResult
            {
                Data = new
                {
                    Id = 1,
                    Name = "Test 1"
                },
                JsonRequestBehavior = JsonRequestBehavior.AllowGet,
                Settings = { ReferenceLoopHandling = ReferenceLoopHandling.Ignore }
            };
        }
```

طراحی آن با توجه به ارث بری از JsonResult اصلی، مشابه نمونه‌ای است که هم اکنون از آن استفاده می‌کنید. فقط اینبار قابلیت تنظیم Settings پیشرفته‌ای نیز به آن اضافه شده‌است.

تا اینجا قسمت ارسال اطلاعات از سمت سرور به سمت کاربر بازنویسی شد. امکان بازنویسی و تعویض موتور پردازش JSON دریافتی از سمت کاربر، در سمت سرور نیز وجود دارد. خود ASP.NET MVC به صورت استاندارد توسط کلاسی به نام JsonValueProviderFactory ، اطلاعات اشیاء JSON دریافتی از سمت کاربر را پردازش می‌کند. در اینجا نیز اگر دقت کنید از کلاس JavaScriptSerializer استفاده شده‌است.
برای جایگزینی آن باید یک ValueProvider جدید را تهیه کنیم:

```
using System;
using System.Dynamic;
using System.Globalization;
using System.IO;
using System.Web.Mvc;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;

namespace MvcJsonNetTests.Utils
{
    public class JsonNetValueProviderFactory : ValueProviderFactory
    {
        public override IValueProvider GetValueProvider(ControllerContext controllerContext)
        {
            if (controllerContext == null)
                throw new ArgumentNullException("controllerContext");

            if (controllerContext.HttpContext == null ||
                controllerContext.HttpContext.Request == null ||
                controllerContext.HttpContext.Request.ContentType == null)
            {
                return null;
            }

            if (!controllerContext.HttpContext.Request.ContentType.StartsWith(
                    "application/json", StringComparison.OrdinalIgnoreCase))
            {
                return null;
            }

            using (var reader = new StreamReader(controllerContext.HttpContext.Request.InputStream))
            {
                var bodyText = reader.ReadToEnd();
```

```
                return string.IsNullOrEmpty(bodyText)
                    ? null
                    : new DictionaryValueProvider<object>(
                        JsonConvert.DeserializeObject<ExpandoObject>(bodyText, new
JsonSerializerSettings
                        {
                            Converters = { new ExpandoObjectConverter() }
                        }),
                        CultureInfo.CurrentCulture);
            }
        }
    }
}
```

در اینجا ابتدا بررسی می‌شود که آیا اطلاعات دریافتی دارای هدر application/json است یا خیر. اگر خیر، توسط این کلاس پردازش نخواهند شد.
در ادامه، اطلاعات JSON دریافتی به شکل یک رشته‌ی خام دریافت شده و سپس به متد JsonConvert.DeserializeObject ارسال می‌شود. با استفاده از تنظیم ExpandoObjectConverter، می‌توان محدودیت کلاس JavaScriptSerializer را در مورد خواص و یا پارامترهای dynamic، برطرف کرد.

```
[HttpPost]
  public ActionResult TestValueProvider(string data1, dynamic data2)
```

برای مثال اینبار می‌توان اطلاعات دریافتی را همانند امضای متد فوق، به یک پارامتر از نوع dynamic، بدون مشکل نگاشت کرد.

و در آخر برای معرفی این ValueProvider جدید می‌توان در فایل Global.asax.cs به نحو ذیل عمل نمود:

```
using System.Linq;
using System.Web.Mvc;
using System.Web.Routing;
using MvcJsonNetTests.Utils;

namespace MvcJsonNetTests
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ValueProviderFactories.Factories.Remove(
                ValueProviderFactories.Factories.OfType<JsonValueProviderFactory>().FirstOrDefault());
            ValueProviderFactories.Factories.Add(new JsonNetValueProviderFactory());
        }
    }
}
```

ابتدا نمونه‌ی قدیمی آن یعنی JsonValueProviderFactory حذف می‌شود و سپس نمونه‌ی جدیدی که از JSON.NET استفاده می‌کند، معرفی خواهد شد.

البته نگارش بعدی ASP.NET MVC موتور پردازشی JSON خود را از طریق تزریق وابستگی‌ها دریافت می‌کند و از همان ابتدای کار قابل تنظیم و تعویض است. مقدار پیش فرض آن نیز به JSON.NET تنظیم شده‌است.

دریافت یک مثال کامل

MvcJsonNetTests.zip

نظرات خوانندگان

نویسنده: مهدی پایروند
تاریخ: ۱۳۹۳/۰۶/۱۶ ۱۱:۱۸

تا جایی که من میدونم JavaScriptSerializer با Dictionaryها هم مشکل داشت ولی JSON.NET این مشکل رو نداره.

نویسنده: احمد
تاریخ: ۱۳۹۳/۰۶/۱۶ ۱۳:۲۲

سلام. اگر در مثال پیوست شده کلاس زیر را استفاده کنیم خطا می‌دهد:

```
public class MyClass
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }


 [HttpPost]
        public ActionResult TestValueProvider(string data1, MyClass data2)
        {

            var id = data2.Id;
            var name = data2.Name;

            return new JsonNetResult
            {
                Data = new { result = data1 },
                JsonRequestBehavior = JsonRequestBehavior.AllowGet,
                Settings = { ReferenceLoopHandling = ReferenceLoopHandling.Ignore }
            };
        }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۶ ۱۵:۱۷

نسخه‌ی بهبود یافته JsonNetValueProviderFactory را در اینجا می‌توانید مطالعه کنید. نسخه‌ی JsonNetResult آن جالب نیست چون از string استفاده کرده بجای stream .

JsonNetValueProviderFactory.cs

+ نحوه‌ی ثبت بهتر این کلاس دقیقا در همان ایندکس اصلی آن:

```
public static void RegisterFactory()
        {
            var defaultJsonFactory = ValueProviderFactories.Factories
                .OfType<JsonValueProviderFactory>().FirstOrDefault();
            var index = ValueProviderFactories.Factories.IndexOf(defaultJsonFactory);
            ValueProviderFactories.Factories.Remove(defaultJsonFactory);
            ValueProviderFactories.Factories.Insert(index, new JsonNetValueProviderFactory());
        }
```

نویسنده: احمد
تاریخ: ۱۳۹۳/۰۶/۱۶ ۱۵:۵۱

خیلی ممنون.
یک مشکل دیگر:

```
public enum MyEnum
    {
        One=1,
        Two=2
    }

 [HttpPost]
```

```
 public ActionResult TestValueProvider(string data1,MyEnum id,string name)
{
```

Enum قابل تبدیل نیست و خطای مشابه سوال قبل را می‌دهد.

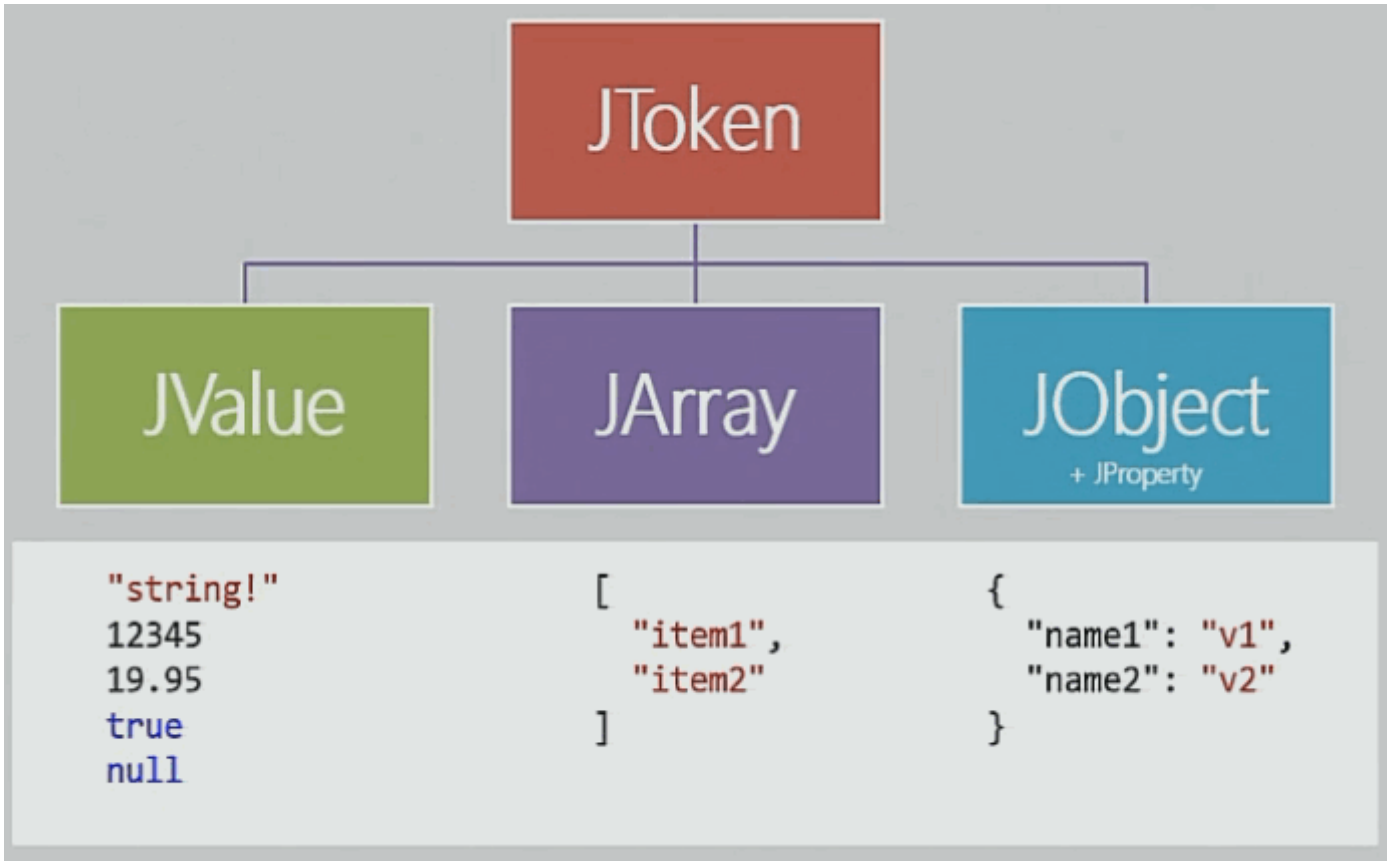
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۶ ۱۸:۵۱

مشکلی نیست. enum در سمت کلاینت باید به صورت رشته‌ای مقدار دهی شود:

```
$.ajax({
                //....
                data: JSON.stringify(
                {
                    data1: "Test1",
                    data2: { Id: 1, Name: "dynamic test" },
                    data3: 'Two' // مقدار دهی عضو ایی‌نام
                }),
```

عنوان: **LINQ to JSON به کمک JSON.NET**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۹ ۲۳:۵۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, MVC, JSON, ASP.NET Web API, json.net

عموما از امکانات LINQ to JSON کتابخانه‌ی JSON.NET زمانی استفاده می‌شود که ورودی JSON تو در توی حجیمی را دریافت کرده‌اید اما قصد ندارید به ازای تمام موجودیت‌های آن یک کلاس معادل را جهت نگاشت به آن‌ها تهیه کنید و صرفا یک یا چند مقدار تو در توی آن جهت عملیات استخراج نهایی مدنظر است. به علاوه در اینجا LINQ to JSON واژه‌ی کلیدی dynamic را نیز پشتیبانی می‌کند.



همانطور که در تصویر مشخص است، خروجی‌های JSON عموما ترکیبی هستند از مقادیر، آرایه‌ها و اشیاء. هر کدام از این‌ها در LINQ to JSON به اشیاء JValue، JArray و JObject نگاشت می‌شوند. البته در حالت JObject هر عضو به یک JProperty و JValue تجزیه خواهد شد.
برای مثال آرایه [1,2] تشکیل شده‌است از یک JArray به همراه دو JValue که مقادیر آن‌را تشکیل می‌دهند. اگر مستقیما بخواهیم یک JArray را تشکیل دهیم می‌توان از شیء JArray استفاده کرد:

```
var array = new JArray(1, 2, 3);
var arrayToJson = array.ToString();
```

و اگر یک JSON رشته‌ای دریافتی را داریم می‌توان از متد Parse مربوط به JArray کمک گرفت:

```
var json = "[1,2,3]";
var jArray= JArray.Parse(json);
var val = (int)jArray[0];
```

خروجی JArray یک لیست از JTokenها است و با آن می‌توان مانند لیست‌های معمولی کار کرد.

در حالت کار با اشیاء، شیء JObject امکان تهیه اشیاء JSON ایی را دارا است که می‌تواند مجموعه‌ای از JPropertyها باشد:

```
var jObject = new JObject(
new JProperty("prop1", "value1"),
new JProperty("prop2", "value2")
);
var jObjectToJson = jObject.ToString();
```

با JObject به صورت dynamic نیز می‌توان کار کرد:

```
dynamic jObj = new JObject();
jObj.Prop1 = "value1";
jObj.Prop2 = "value2";
jObj.Roles = new[] {"Admin", "User"};
```

این روش بسیار شبیه است به حالتی که با اشیاء جاوا اسکریپتی در سمت کلاینت می‌توان کار کرد.
و حالت عکس آن توسط متد JObject.Parse قابل انجام است:

```
var json = "{ 'prop1': 'value1', 'prop2': 'value2'}";
var jObj = JObject.Parse(json);
var val1 = (string)jObj["prop1"];
```

اکنون که با اجزای تشکیل دهنده‌ی LINQ to JSON آشنا شدیم، مثال ذیل را درنظر بگیرید:

```
var array = @"[
{
  'prop1': 'value1',
  'prop2': 'value2'
},
{
  'prop1': 'test1',
  'prop2': 'test2'
}
]";
var objects = JArray.Parse(array);
var obj1 = objects.FirstOrDefault(token => (string) token["prop1"] == "value1");
```

خروجی JArray یا JObject از نوع IEnumerable است و بر روی آن‌ها می‌توان کلیه متدهای LINQ را فراخوانی کرد. برای مثال در اینجا اولین شیء‌ایی که مقدار خاصیت prop1 آن مساوی value1 است، یافت می‌شود و یا می‌توان اشیاء را بر اساس مقدار خاصیتی مرتب کرده و سپس آن‌ها را بازگشت داد:

```
var values = objects.OrderBy(token => (string) token["prop1"])
.Select(token => new {Value = (string) token["prop2"]})
.ToList();
```

امکان انجام sub queries نیز در اینجا پیش بینی شده‌است:

```
var array = @"[
{
  'prop1': 'value1',
  'prop2': [1,2]
},
{
  'prop1': 'test1',
  'prop2': [1,2,3]
}
]";
var objects = JArray.Parse(array);
var objectContaining3 = objects.Where(token => token["prop2"].Any(v => (int)v == 3)).ToList();
```

در این مثال، خواص prop2 از نوع آرایه‌ای از اعداد صحیح هستند. با کوئری نوشته شده، اشیایی که خاصیت prop2 آن‌ها دارای عضو 3 است، یافت می‌شوند.

عنوان: **تنظیمات JSON در ASP.NET Web API**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۱ ۱۳:۵۵
آدرس: www.dotnettips.info
گروه‌ها: JSON, ASP.NET Web API, json.net

ASP.NET Web API در سمت سرور، برای مدیریت ApiControllerها و در سمت کلاینت‌های دات نتی آن، برای مدیریت HttpClient، به صورت پیش فرض از JSON.NET استفاده می‌کند. در ادامه نگاهی خواهیم داشت به تنظیمات JSON در سرور و کلاینت‌های ASP.NET Web API.

آماده سازی یک مثال Self host

برای اینکه خروجی‌های JSON را بهتر و بدون نیاز به ابزار خاصی مشاهده کنیم، می‌توان یک پروژه‌ی کنسول جدید را آغاز کرده و سپس آن‌را تبدیل به Host مخصوص Web API کرد. برای اینکار تنها کافی است در کنسول پاور شل نیوگت دستور ذیل را صادر کنید:

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

سپس کنترلر Web API ما از کدهای ذیل تشکیل خواهد شد که در آن در متد Post، قصد داریم اصل محتوای دریافتی از کاربر را نمایش دهیم. توسط متد GetAll آن، خروجی نهایی JSON آن در سمت کاربر بررسی خواهد شد.

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;

namespace WebApiSelfHostTests
{
    public class UsersController : ApiController
    {
        public IEnumerable<User> GetAllUsers()
        {
            return new[]
            {
                new User{ Id = 1, Name = "User 1", Type = UserType.Admin },
                new User{ Id = 2, Name = "User 2", Type = UserType.User }
            };
        }

        public async Task<HttpResponseMessage> Post(HttpRequestMessage request)
        {
            var jsonContent = await request.Content.ReadAsStringAsync();
            Console.WriteLine("JsonContent (Server Side): {0}", jsonContent);
            return new HttpResponseMessage(HttpStatusCode.Created);
        }
    }
}
```

که در آن شیء کاربر چنین ساختاری را دارد:

```
namespace WebApiSelfHostTests
{
    public enum UserType
    {
        User,
        Admin,
        Writer
    }

    public class User
    {
        public int Id { set; get; }
        public string Name { set; get; }
        public UserType Type { set; get; }
```

```
    }
}
```

برای اعمال تنظیمات self host ابتدا نیاز است یک کلاس Startup مخصوص Owin را تهیه کرد:

```
using System.Web.Http;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;
using Owin;

namespace WebApiSelfHostTests
{
    /// <summary>
    /// PM> Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
    /// </summary>
    public class Startup
    {
        public void Configuration(IAppBuilder appBuilder)
        {
            var config = new HttpConfiguration();
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
                );

            appBuilder.UseWebApi(config);
        }
    }
}
```

که سپس با فراخوانی چند سطر ذیل، سبب راه اندازی سرور Web API، بدون نیاز به IIS خواهد شد:

```
var server = WebApp.Start<Startup>(url: BaseAddress);


Console.WriteLine("Press Enter to quit.");
Console.ReadLine();
server.Dispose();
```

در ادامه اگر در سمت کلاینت، دستورات ذیل را برای دریافت لیست کاربران صادر کنیم:

```
using (var client = new HttpClient())
{
   var response = client.GetAsync(BaseAddress + "api/users").Result;
   Console.WriteLine("Response: {0}", response);
   Console.WriteLine("JsonContent (Client Side): {0}", response.Content.ReadAsStringAsync().Result);
}
```

به این خروجی خواهیم رسید:

```
JsonContent (Client Side): [{"Id":1,"Name":"User 1","Type":1},{"Id":2,"Name":"User 2","Type":0}]
```

همانطور که ملاحظه می‌کنید، مقدار Type مساوی صفر است. در اینجا چون Type را به صورت enum تعریف کرده‌ایم، به صورت پیش فرض مقدار عددی عضو انتخابی در JSON نهایی درج می‌گردد.

تنظیمات JSON سمت سرور Web API

برای تغییر این خروجی، در سمت سرور تنها کافی است به کلاس Startup مراجعه و HttpConfiguration را به صورت ذیل تنظیم کنیم:

```
public class Startup
```

```
    {
        public void Configuration(IAppBuilder appBuilder)
        {
            var config = new HttpConfiguration();
            config.Formatters.JsonFormatter.SerializerSettings = new JsonSerializerSettings
            {
                Converters = { new StringEnumConverter() }
            };
```

در اینجا با انتخاب StringEnumConverter، سبب خواهیم شد تا کلیه مقادیر enum، دقیقا مساوی همان مقدار اصلی رشته‌ای آن‌ها در JSON نهایی درج شوند.
اینبار اگر برنامه را اجرا کنیم، چنین خروجی حاصل می‌گردد و در آن دیگر Type مساوی صفر نیست:

```
JsonContent (Client Side): [{"Id":1,"Name":"User 1","Type":"Admin"},{"Id":2,"Name":"User
2","Type":"User"}]
```

تنظیمات JSON سمت کلاینت Web API

اکنون در سمت کلاینت قصد داریم اطلاعات یک کاربر را با فرمت JSON به سمت سرور ارسال کنیم. روش متداول آن توسط کتابخانه‌ی HttpClient، استفاده از متد PostAsJsonAsync است:

```
var user = new User
{
   Id = 1,
   Name = "User 1",
   Type = UserType.Writer
};

var client = new HttpClient();
client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

var response = client.PostAsJsonAsync(BaseAddress + "api/users", user).Result;
Console.WriteLine("Response: {0}", response);
```

با این خروجی سمت سرور

```
JsonContent (Server Side): {"Id":1,"Name":"User 1","Type":2}
```

در اینجا نیز Type به صورت عددی ارسال شده‌است. برای تغییر آن نیاز است به متدی با سطح پایین‌تر از PostAsJsonAsync مراجعه کنیم تا در آن بتوان JsonMediaTypeFormatter را مقدار دهی کرد:

```
var jsonMediaTypeFormatter = new JsonMediaTypeFormatter
            {
                SerializerSettings = new JsonSerializerSettings
                {
                    Converters = { new StringEnumConverter() }
                }
            };
            var response = client.PostAsync(BaseAddress + "api/users", user,
jsonMediaTypeFormatter).Result;
            Console.WriteLine("Response: {0}", response);
```

خاصیت SerializerSettings کلاس JsonMediaTypeFormatter برای اعمال تنظیمات JSON.NET پیش بینی شده‌است.
اینبار مقدار دریافتی در سمت سرور به صورت ذیل است و در آن، Type دیگر عددی نیست:

```
JsonContent (Server Side): {"Id":1,"Name":"User 1","Type":"Writer"}
```

مثال کامل این بحث را از اینجا می‌توانید دریافت کنید:

UsersController.zip

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۴ ۱۰:۴۵

یک نکته‌ی تکمیلی

اگر نمی‌خواهید یک وابستگی جدید را (Microsoft.AspNet.WebApi.Client) به پروژه اضافه کنید، کدهای ذیل همان کار HttpClient را برای ارسال اطلاعات، انجام می‌دهند. کلاس WebRequest آن در فضای نام System.Net موجود است :

```
using System;
using System.IO;
using System.Net;
using Newtonsoft.Json;

namespace WebToolkit
{
    public class SimpleHttp
    {
        public HttpStatusCode PostAsJson(string url, object data, JsonSerializerSettings settings)
        {
            if (string.IsNullOrWhiteSpace(url))
                throw new ArgumentNullException("url");

            return PostAsJson(new Uri(url), data, settings);
        }

        public HttpStatusCode PostAsJson(Uri url, object data, JsonSerializerSettings settings)
        {
            if (url == null)
                throw new ArgumentNullException("url");

            var postRequest = (HttpWebRequest)WebRequest.Create(url);
            postRequest.Method = "POST";
            postRequest.UserAgent = "SimpleHttp/1.0";
            postRequest.ContentType = "application/json; charset=utf-8";

            using (var stream = new StreamWriter(postRequest.GetRequestStream()))
            {
                var serializer = JsonSerializer.Create(settings);
                using (var writer = new JsonTextWriter(stream))
                {
                    serializer.Serialize(writer, data);
                    writer.Flush();
                }
            }

            using (var response = (HttpWebResponse)postRequest.GetResponse())
            {
                return response.StatusCode;
            }
        }
    }
}
```

نویسنده: رشیدیان
تاریخ: ۱۳۹۳/۰۶/۲۶ ۱۸:۱

سلام و وقت بخیر
من وقتی میخوام اطلاعات یک فایل جیسون رو به آبجکت تبدیل کنم، با این خطا مواجه میشم:

.Additional text encountered after finished reading JSON content: ,. Path '', line 1, position 6982

بعد از جستجو متوجه شدم که خطا به دلیل وجود کرکترهای کنترلی هست، پس فایل مذکور رو با روشهای زیر (هر کدام رو جداگانه تست کردم) تمیز کردم:

```
public string RemoveControlCharacters1(string input)
        {
            string output = Regex.Replace(input, @"[\u0000-\u001F]", string.Empty);
            return output;
```

```
        }

        public string RemoveControlCharacters2(string input)
        {
            string output = new string(input.Where(c => !char.IsControl(c)).ToArray());
            return output;
        }

        public string RemoveControlCharacters3(string inString)
        {
            if (inString == null) return null;
            StringBuilder newString = new StringBuilder();
            char ch;
            for (int i = 0; i < inString.Length; i++)
            {
                ch = inString[i];
                if (!char.IsControl(ch))
                {
                    newString.Append(ch);
                }
            }
            return newString.ToString();
        }
```

اما کماکان همان خطا را در زمان اجر میبینم.
آیا مشکل چیز دیگری است؟
پرسش: چطور میشود به جیسون دات نت گفت که اصلا کرکترهای کنترلی و یا چیزهایی را که ممکن است خطا ایجاد کنند، ندید بگیرد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۶ ۱۸:۱۲

با تنظیم eventArgs.ErrorContext.Handled = true از خطاهای موجود صرفنظر می‌شود:

```
new JsonSerializerSettings
{
  Error = (sender, eventArgs) =>
  {
      Debug.WriteLine(eventArgs.ErrorContext.Error.Message);
      //if an error happens we can mark it as handled, and it will continue
      eventArgs.ErrorContext.Handled = true;
   }
}
```

نویسنده: رشیدیان
تاریخ: ۱۳۹۳/۰۶/۲۶ ۱۸:۲۳

سپاسگزارم از پاسخ سریع شما.
ببخشید کد من به این شکل هست و نمیدونم کجا باید تغییرات رو اعمال کنم:

```
var  items = JsonConvert.DeserializeObject<List<Classified>>(json);
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۶ ۱۸:۲۶

در پارامتر دوم متد « تبدیل JSON رشته‌ای به اشیاء دات نت ».

نویسنده: رضایی
تاریخ: ۱۳۹۳/۰۶/۲۶ ۱۸:۴۶

سلام؛ من از کد زیر استفاده کردم

```
myUserApi.Id = UserId;
```

```
                return new JsonNetResult
                {
                    Data = myUserApi,
                    ContentType = "application/json"
                };
```

اما این خروجی تولید میشه

```
{
  "$id": "1",
  "Settings": {
    "$id": "2",
    "ReferenceLoopHandling": 0,
    "MissingMemberHandling": 0,
    "ObjectCreationHandling": 0,
    "NullValueHandling": 0,
    "DefaultValueHandling": 0,
    "Converters": [],
    "PreserveReferencesHandling": 0,
    "TypeNameHandling": 0,
    "MetadataPropertyHandling": 0,
    "TypeNameAssemblyFormat": 0,
    "ConstructorHandling": 0,
    "ContractResolver": null,
    "ReferenceResolver": null,
    "TraceWriter": null,
    "Binder": null,
    "Error": null,
    "Context": {
      "$id": "3",
      "m_additionalContext": null,
      "m_state": 0
    },
    "DateFormatString": "yyyy'-'MM'-'dd'T'HH':'mm':'ss.FFFFFFFK",
    "MaxDepth": null,
    "Formatting": 0,
    "DateFormatHandling": 0,
    "DateTimeZoneHandling": 3,
    "DateParseHandling": 1,
    "FloatFormatHandling": 0,
    "FloatParseHandling": 0,
    "StringEscapeHandling": 0,
    "Culture": "(Default)",
    "CheckAdditionalContent": false
  },
  "ContentEncoding": null,
  "ContentType": "application/json",
  "Data": "{\"Id\":2}",
  "JsonRequestBehavior": 1,
  "MaxJsonLength": null,
  "RecursionLimit": null
}
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۲۶ ۱۸:۵۵

مرتبط است به نکته‌ی « تهیه خروجی JSON از مدل‌های مرتبط، بدون Stack overflow »