

## 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 نویسی 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 نت 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 در نظر 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 نت 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 JsonConvert، 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 نت، 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 تویی 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 استاندارد 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 کارایی 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 نهایتاً GC کار کمتری را باید انجام دهد.

### تبدیل رشته‌ای به اشیاء دات نت

اگر رشته‌ی jsonData ای 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 شده است. 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 صرفاً 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. مثلاً 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 نت پیروی نمی‌کنند، در طراحی خود استفاده می‌کنند. در اینجا می‌توان نام خاصیت نهایی را که قرار است رندر شود نیز صریحاً 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 صرفاً 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:

۲۱:۵۴ ۱۳۹۳/۰۶/۱۳

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:

۲۱:۵۹ ۱۳۹۳/۰۶/۱۳

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:

۱۴:۳ ۱۳۹۳/۰۶/۱۴

"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:

۱۴:۱۹ ۱۳۹۳/۰۶/۱۴

- 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 دقیقاً از JSON.NET استفاده می‌کند.

نویسنده:

وحید نصیری

تاریخ:

۱۲:۳۵ ۱۳۹۳/۰۷/۱۸

## 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);
            }
        }
    }
}

```

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

## گرفتن خروجی از CamelCase در 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 CamelCasePropertyNameContractResolver 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 در نظر 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: ۱۵:۱۷ ۱۳۹۳/۰۶/۲۳

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()
});
```

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، مستقیماً در `response.Output` استریم 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 شده است. کدهای ابتدای متد `ExecuteResult` با [کدهای اصلی](#) `JsonResult` یکی هستند. فقط انتهای کار بجای استفاده از `JavaScriptSerializer`، از `JSON.NET` استفاده شده است. در این حالت برای استفاده از این `Action Result` جدید می توان 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 نمونه ای است که هم اکنون از آن استفاده می کنید. فقط اینبار قابلیت تنظیم `Settings` پیشرفته ای نیز به آن اضافه شده است.

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 می کند. در اینجا نیز اگر دقت کنید از کلاس `JavaScriptSerializer` استفاده شده است. 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: ۱۱:۱۸ ۱۳۹۳/۰۶/۱۶

تا جایی که من میدونم [JavaScriptSerializer](#) با Dictionary ها هم مشکل داشت ولی JSON.NET این مشکل رو نداره.

نویسنده: احمد  
تاریخ: ۱۳:۲۲ ۱۳۹۳/۰۶/۱۶

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

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

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 دقیقاً 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: ۱۵:۵۱ ۱۳۹۳/۰۶/۱۶

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

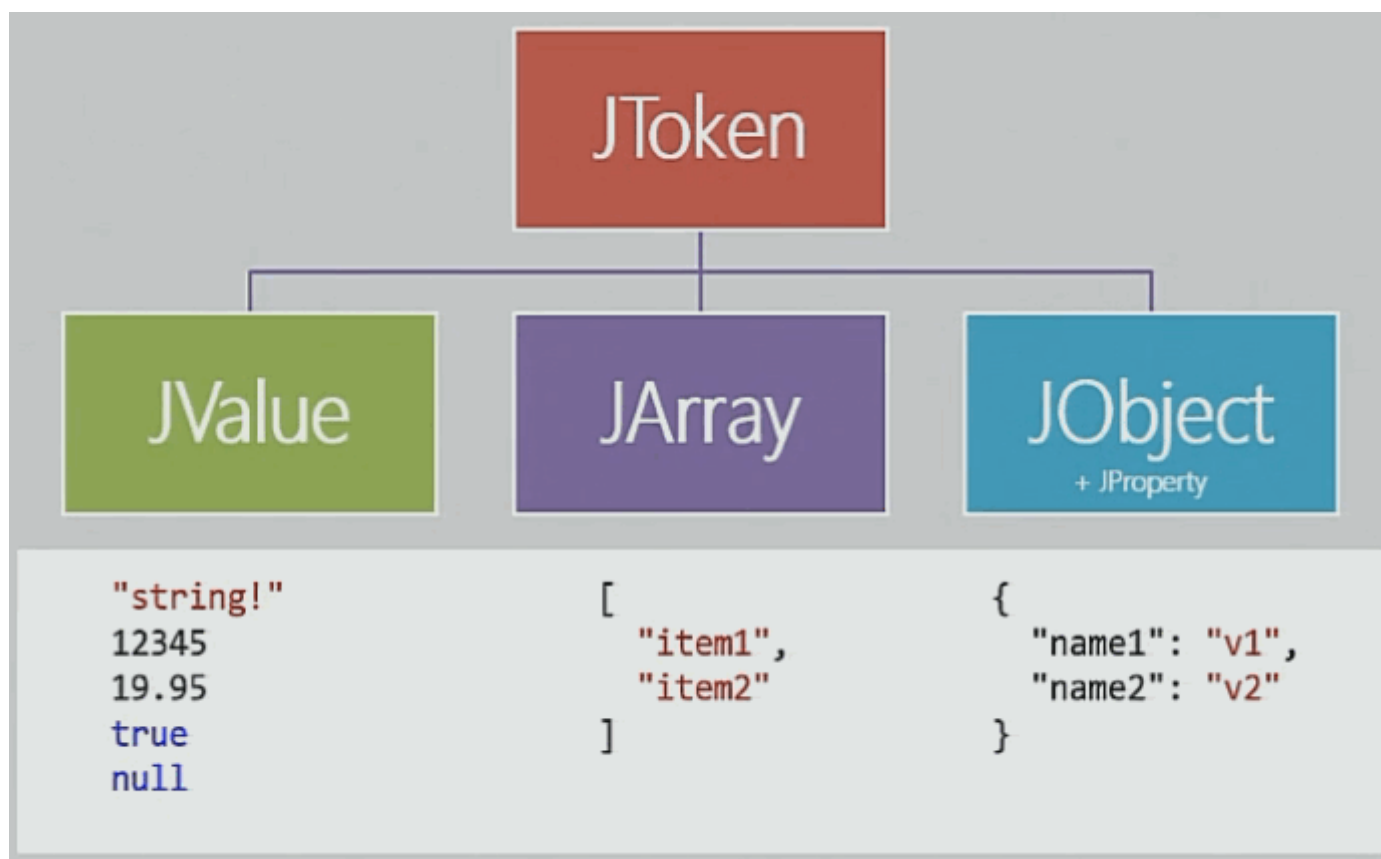
تاریخ:

۱۸:۵۱ ۱۳۹۳/۰۶/۱۶

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 ای‌نام
    }
    ),
```

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 صرفاً 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 عموماً 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 مستقیماً 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 JToken ها 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 JProperty ها 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 در نظر 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.

ASP.NET Web API در سمت سرور، برای مدیریت ApiController ها 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`، دقیقاً 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 پیش‌بینی 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:



## نظرات خوانندگان

نویسنده:

وحید نصیری

تاریخ:

۱۰:۴۵ ۱۳۹۳/۰۶/۲۴

## 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.IsNullOrEmpty(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:

۱۸:۱ ۱۳۹۳/۰۶/۲۶

## 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: ۱۸:۱۲ ۱۳۹۳/۰۶/۲۶

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: ۱۸:۲۳ ۱۳۹۳/۰۶/۲۶

سپاسگزارم از پاسخ سریع شما.

بخشید 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: ۱۸:۲۶ ۱۳۹۳/۰۶/۲۶

در پارامتر دوم متد «[تبدیل JSON رشته‌ای به اشیاء دات نت](#)».

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

سلام! من از کد زیر استفاده کردم

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

```
return new JsonResult
{
    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.FFFFFFFF",
    "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: ۱۸:۵۵ ۱۳۹۳/۰۶/۲۶

مرتبط است به نکته‌ی « [تهیه خروجی JSON از مدل‌های مرتبط، بدون Stack overflow](#) »

Kendo UI DataSource جهت تامین داده‌های سمت کلاینت [ویجت‌های مختلف KendoUI](#) طراحی شده‌است و به عنوان یک اینترفیس استاندارد قابل استفاده توسط تمام کنترل‌های داده‌ای Kendo UI کاربرد دارد. Kendo UI DataSource امکان کار با منابع داده محلی، مانند اشیاء و آرایه‌های جاوا اسکریپتی و همچنین منابع تامین شده از راه دور، مانند JSONP، JSON و XML را دارد. به علاوه توسط آن می‌توان اعمال ثبت، ویرایش و حذف اطلاعات، به همراه صفحه بندی، گروه بندی و مرتب سازی داده‌ها را کنترل کرد.

### استفاده از منابع داده محلی

در ادامه مثالی را از نحوه‌ی استفاده از یک منبع داده محلی جاوا اسکریپتی، مشاهده می‌کنید:

```
<script type="text/javascript">
$(function () {
    var cars = [
        { "Year": 2000, "Make": "Hyundai", "Model": "Elantra" },
        { "Year": 2001, "Make": "Hyundai", "Model": "Sonata" },
        { "Year": 2002, "Make": "Toyota", "Model": "Corolla" },
        { "Year": 2003, "Make": "Toyota", "Model": "Yaris" },
        { "Year": 2004, "Make": "Honda", "Model": "CRV" },
        { "Year": 2005, "Make": "Honda", "Model": "Accord" },
        { "Year": 2000, "Make": "Honda", "Model": "Accord" },
        { "Year": 2002, "Make": "Kia", "Model": "Sedona" },
        { "Year": 2004, "Make": "Fiat", "Model": "One" },
        { "Year": 2005, "Make": "BMW", "Model": "M3" },
        { "Year": 2008, "Make": "BMW", "Model": "X5" }
    ];

    var carsDataSource = new kendo.data.DataSource({
        data: cars
    });

    carsDataSource.read();
    alert(carsDataSource.total());
});
</script>
```

در اینجا cars آرایه‌ای از اشیاء جاوا اسکریپتی بیانگر ساختار یک خودرو است. سپس برای معرفی آن به Kendo UI، کار با مقدار دهی خاصیت data مربوط به new kendo.data.DataSource شروع می‌شود. ذکر new kendo.data.DataSource به تنهایی به معنای مقدار دهی اولیه است و در این حالت منبع داده مورد نظر، استفاده نخواهد شد. برای مثال اگر متد total آن را جهت یافتن تعداد عناصر موجود در آن فراخوانی کنید، صفر را بازگشت می‌دهد. برای شروع به کار با آن، نیاز است ابتدا متد read را بر روی این منبع داده مقدار دهی شده، فراخوانی کرد.

### استفاده از منابع داده راه دور

در برنامه‌های کاربردی، عموماً نیاز است تا منبع داده را از یک وب سرور تامین کرد. در اینجا نحوه‌ی خواندن اطلاعات JSON بازگشت داده شده از جستجوی توئیتر را مشاهده می‌کنید:

```
<script type="text/javascript">
$(function () {
    var twitterDataSource = new kendo.data.DataSource({
        transport: {
            read: {
                url: "http://search.twitter.com/search.json",
                dataType: "jsonp",
                contentType: 'application/json; charset=utf-8',
                type: 'GET',
                data: { q: "#kendoui" }
            },
            schema: { data: "results" }
        }
    });
});
```

```

    },
    error: function (e) {
        alert(e.errorThrown.stack);
    }
  });
});
</script>

```

در قسمت transport، جزئیات تبادل اطلاعات با سرور راه دور مشخص می‌شود؛ برای مثال url ارائه دهنده‌ی سرویس، dataType بیانگر نوع داده مورد انتظار و data کار مقدار دهی پارامتر مورد انتظار توسط سرویس توئیتر را انجام می‌دهد. در اینجا چون صرفاً عملیات خواندن اطلاعات صورت می‌گیرد، خاصیت read مقدار دهی شده‌است. در قسمت schema مشخص می‌کنیم که اطلاعات JSON بازگشت داده شده توسط توئیتر، در فیلد results آن قرار دارد.

### کار با منابع داده OData

علاوه بر فرمت‌های یاد شده، Kendo UI DataSource امکان کار با اطلاعاتی [از نوع OData](#) را نیز دارا است که تنظیمات ابتدایی آن به صورت ذیل است:

```

<script type="text/javascript">
    var moviesDataSource = new kendo.data.DataSource({
        type: "odata",
        transport: {
            read: "http://demos.kendoui.com/service/Northwind.svc/Orders"
        },
        error: function (e) {
            alert(e.errorThrown.stack);
        }
    });
});
</script>

```

همانطور که ملاحظه می‌کنید، تنظیمات ابتدایی آن اندکی با حالت remote data پیشین متفاوت است. در اینجا ابتدا نوع داده‌ی بازگشتی مشخص می‌شود و در قسمت transport، خاصیت read آن، آدرس سرویس را دریافت می‌کند.

### یک مثال: دریافت اطلاعات از ASP.NET Web API

یک پروژه‌ی جدید ASP.NET را آغاز کنید. تفاوتی نمی‌کند که Web forms باشد یا MVC؛ از این جهت که مباحث [Web API](#) در هر دو یکسان است.

سپس یک کنترلر جدید Web API را به نام ProductsController با محتوای زیر ایجاد کنید:

```

using System.Collections.Generic;
using System.Web.Http;

namespace KendoUI02
{
    public class Product
    {
        public int Id { set; get; }
        public string Name { set; get; }
    }

    public class ProductsController : ApiController
    {
        public IEnumerable<Product> Get()
        {
            var products = new List<Product>();
            for (var i = 1; i <= 100; i++)
            {
                products.Add(new Product { Id = i, Name = "Product " + i });
            }
            return products;
        }
    }
}

```

در این مثال، هدف صرفاً ارائه یک خروجی ساده JSON از طرف سرور است.  
در ادامه نیاز است تعریف مسیریابی ذیل نیز به فایل Global.asax.cs برنامه اضافه شود تا بتوان به آدرس api/products سایت، دسترسی یافت:

```
using System;
using System.Web.Http;
using System.Web.Routing;

namespace KendoUI02
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

در ادامه فایلی را به نام Index.html (یا در یک View و یا یک فایل aspx دلخواه)، محتوای ذیل را اضافه کنید:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Kendo UI: Implementing the Grid</title>

    <link href="styles/kendo.common.min.css" rel="stylesheet" type="text/css" />
    <link href="styles/kendo.default.min.css" rel="stylesheet" type="text/css" />
    <script src="js/jquery.min.js" type="text/javascript"></script>
    <script src="js/kendo.all.min.js" type="text/javascript"></script>
</head>
<body>

    <div id="report-grid"></div>
    <script type="text/javascript">
        $(function () {
            var productsDataSource = new kendo.data.DataSource({
                transport: {
                    read: {
                        url: "api/products",
                        dataType: "json",
                        contentType: 'application/json; charset=utf-8',
                        type: 'GET'
                    }
                },
                error: function (e) {
                    alert(e.errorThrown.stack);
                },
                pageSize: 5,
                sort: { field: "Id", dir: "desc" }
            });

            $("#report-grid").kendoGrid({
                dataSource: productsDataSource,
                autoBind: true,
                scrollable: false,
                pageable: true,
                sortable: true,
                columns: [
                    { field: "Id", title: "#" },
                    { field: "Name", title: "Product" }
                ]
            });
        });
    </script>
</body>
</html>
```



- ابتدا فایل‌های اسکریپت و CSS مورد نیاز Kendo UI اضافه شده‌اند.
- گرید صفحه، در محل div ایی با id مساوی report-grid تشکیل خواهد شد.
- سپس DataSource ایی که به آدرس api/products اشاره می‌کند، تعریف شده و در آخر productsDataSource را توسط یک kendoGrid نمایش داده‌ایم.
- نحوه‌ی تعریف productsDataSource، در قسمت استفاده از منابع داده راه دور ابتدای بحث توضیح داده شد. در اینجا فقط دو خاصیت pageSize و sort نیز به آن اضافه شده‌اند. این دو خاصیت بر روی نحوه‌ی نمایش گرید نهایی تاثیر گذار هستند. pageSize تعداد رکورد هر صفحه را مشخص می‌کند و sort نحوه‌ی مرتب سازی را بر اساس فیلد Id و در حالت نزولی قرار می‌دهد.
- در ادامه، ابتدایی‌ترین حالت کار با kendoGrid را ملاحظه می‌کنید.
- تنظیم dataSource و autoBind: true (حالت پیش فرض)، سبب خواهند شد تا به صورت خودکار، اطلاعات JSON از مسیر api/products خوانده شوند.
- سه خاصیت بعدی صفحه بندی و مرتب سازی خودکار ستون‌ها را فعال می‌کنند.
- در آخر هم دو ستون گرید، بر اساس نام‌های خواص کلاس Product تعریف شده‌اند.

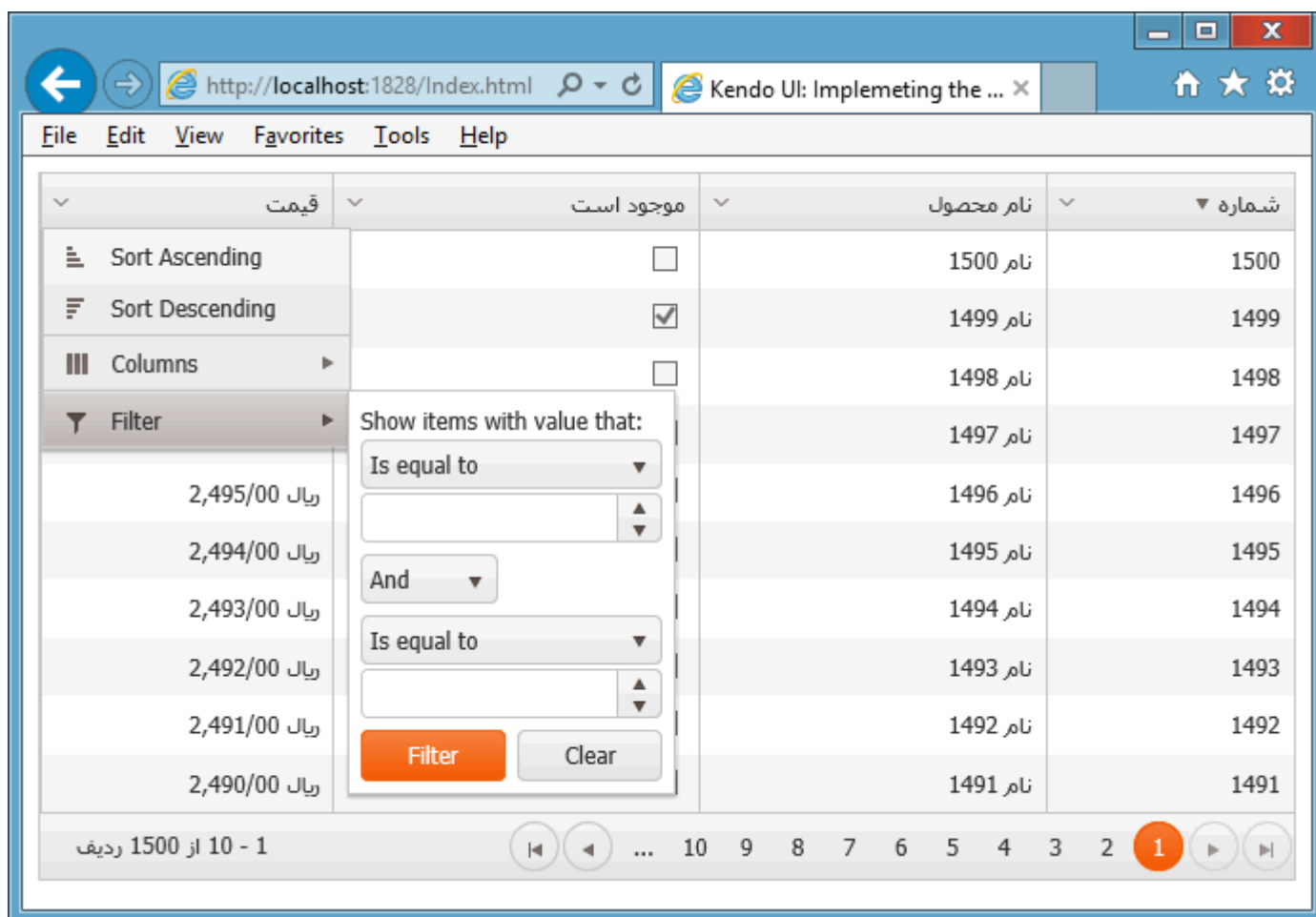
#	Product
5	Product 5
4	Product 4
3	Product 3
2	Product 2
1	Product 1

96 - 100 of 100 items

سورس کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[KendoUI02.zip](#)

پس از آشنایی مقدماتی با [Kendo UI DataSource](#)، اکنون می‌خواهیم از آن جهت صفحه بندی، مرتب سازی و جستجوی پویای سمت سرور استفاده کنیم. در مثال قبلی، هر چند صفحه بندی فعال بود، اما پس از دریافت تمام اطلاعات، این اعمال در سمت کاربر انجام و مدیریت می‌شد.



## مدل برنامه

در اینجا قصد داریم لیستی را با ساختار کلاس Product در اختیار Kendo UI گرید قرار دهیم:

```
namespace KendoUI03.Models
{
    public class Product
    {
        public int Id { set; get; }
        public string Name { set; get; }
        public decimal Price { set; get; }
        public bool IsAvailable { set; get; }
    }
}
```

## پیشنیاز تامین داده مخصوص Kendo UI Grid

برای ارائه اطلاعات مخصوص Kendo UI Grid، ابتدا باید در نظر داشت که این گرید، درخواست های صفحه بندی خود را با فرمت ذیل ارسال می کند. همانطور که مشاهده می کنید، صرفاً یک کوئری استرینگ با فرمت JSON را دریافت خواهیم کرد:

```
/api/products?{"take":10,"skip":0,"page":1,"pageSize":10,"sort":[{"field":"Id","dir":"desc"}]}
```

سپس این گرید نیاز به سه فیلد، در خروجی JSON نهایی خواهد داشت:

```
{
  "Data":
  [
    {"Id":1500,"Name":"1500 نام","Price":2499.0,"IsAvailable":false},
    {"Id":1499,"Name":"1499 نام","Price":2498.0,"IsAvailable":true}
  ],
  "Total":1500,
  "Aggregates":null
}
```

فیلد Data که رکوردهای گرید را تامین می کند. فیلد Total که بیانگر تعداد کل رکوردها است و Aggregates که برای گروه بندی بکار می رود.

می توان برای تمام این ها، کلاس و Parser تهیه کرد و یا ... پروژه های سورس بازی به نام [Kendo.DynamicLinq](#) نیز چنین کاری را میسر می سازد که در ادامه از آن استفاده خواهیم کرد. برای نصب آن تنها کافی است دستور ذیل را صادر کنید:

```
PM> Install-Package Kendo.DynamicLinq
```

Kendo.DynamicLinq به صورت خودکار [System.Linq.Dynamic](#) را نیز نصب می کند که از آن جهت صفحه بندی پویا استفاده خواهد شد.

## تامین کننده ی داده سمت سرور

همانند مطلب کار با [Kendo UI DataSource](#)، یک ASP.NET Web API Controller جدید را به پروژه اضافه کنید و همچنین مسیریابی های مخصوص آن را به فایل global.asax.cs نیز اضافه نمایید.

```
using System.Linq;
using System.Net.Http;
using System.Web.Http;
using Kendo.DynamicLinq;
using KendoUI03.Models;
using Newtonsoft.Json;

namespace KendoUI03.Controllers
{
    public class ProductsController : ApiController
    {
        public DataSourceResult Get(HttpRequestMessage requestMessage)
        {
            var request = JsonConvert.DeserializeObject<DataSourceRequest>(
                requestMessage.RequestUri.ParseQueryString().GetKey(0)
            );

            var list = ProductDataSource.LatestProducts;
            return list.AsQueryable()
                .ToDataSourceResult(request.Take, request.Skip, request.Sort, request.Filter);
        }
    }
}
```

تمام کدهای این کنترلر همین چند سطر فوق هستند. با توجه به ساختار کوئری استرینگی که در ابتدای بحث عنوان شد، نیاز است آنرا توسط کتابخانه‌ی [JSON.NET](#) تبدیل به یک نمونه از [DataSourceRequest](#) نمائیم. این کلاس در Kendo.DynamicLinq تعریف شده است و حاوی اطلاعاتی مانند take و skip کوئری LINQ نهایی است.

ProductDataSource.LatestProducts صرفاً یک لیست جنریک تهیه شده از کلاس Product است. در نهایت با استفاده از متد الحاقی جدید [ToDataSourceResult](#)، به صورت خودکار مباحث صفحه بندی سمت سرور به همراه مرتب سازی اطلاعات، صورت گرفته و اطلاعات نهایی با فرمت [DataSourceResult](#) بازگشت داده می‌شود. DataSourceResult نیز در Kendo.DynamicLinq تعریف شده و سه فیلد یاد شده‌ی Data، Total و Aggregates را تولید می‌کند.

تا اینجا کارهای سمت سرور این مثال به پایان می‌رسد.

## تهیه View نمایش اطلاعات ارسالی از سمت سرور

### اعمال مباحث بومی سازی

```
<head>
  <meta charset="utf-8" />
  <meta http-equiv="Content-Language" content="fa" />
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

  <title>Kendo UI: Implementing the Grid</title>

  <link href="styles/kendo.common.min.css" rel="stylesheet" type="text/css" />
  <!-- شیوه نامی مخصوص راست به چپ سازی -->
  <link href="styles/kendo.rtl.min.css" rel="stylesheet" />
  <link href="styles/kendo.default.min.css" rel="stylesheet" type="text/css" />
  <script src="js/jquery.min.js" type="text/javascript"></script>
  <script src="js/kendo.all.min.js" type="text/javascript"></script>

  <!-- محل سفارشی سازی پیام‌ها و مسایل بومی -->
  <script src="js/cultures/kendo.culture.fa-IR.js" type="text/javascript"></script>
  <script src="js/cultures/kendo.culture.fa.js" type="text/javascript"></script>
  <script src="js/messages/kendo.messages.en-US.js" type="text/javascript"></script>

  <style type="text/css">
    body {
      font-family: tahoma;
      font-size: 9pt;
    }
  </style>

  <script type="text/javascript">
    // جهت استفاده از فایل kendo.culture.fa-IR.js
    kendo.culture("fa-IR");
  </script>
</head>
```

- در اینجا چند فایل js و css جدید اضافه شده‌اند. فایل kendo.rtl.min.css جهت تامین مباحث RTL توکار Kendo UI کاربرد دارد.

- سپس سه فایل kendo.culture.fa.js، kendo.culture.fa-IR.js و kendo.messages.en-US.js نیز اضافه شده‌اند. فایل‌های fa و fa-IR آن‌ها هر چند به ظاهر برای ایران طراحی شده‌اند، اما نام ماه‌های موجود در آن عربی است که نیاز به ویرایش دارد. به همین جهت به سؤرس این فایل‌ها، جهت ویرایش نهایی نیاز خواهد بود که در پوشه‌ی src\js\cultures مجموعه‌ی اصلی Kendo UI موجود هستند (ر.ک. فایل پیوست).

- فایل kendo.messages.en-US.js حاوی تمام پیام‌های مرتبط با Kendo UI است. برای مثال «رکوردهای 10 تا 15 از 1000 ردیف» را در اینجا می‌توانید به فارسی ترجمه کنید.

- متد kendo.culture کار مشخص سازی فرهنگ بومی برنامه را به عهده دارد. برای مثال در اینجا به fa-IR تنظیم شده‌است. این مورد سبب خواهد شد تا از فایل kendo.culture.fa-IR.js استفاده گردد. اگر مقدار آن‌را به fa تنظیم کنید، از فایل kendo.culture.fa.js کمک گرفته خواهد شد.

### راست به چپ سازی گرید

تنها کاری که برای راست به چپ سازی Kendo UI Grid باید صورت گیرد، محصور سازی div آن در یک div با کلاس مساوی k-

rtl است:

```
<div class="k-rtl">
  <div id="report-grid"></div>
</div>
```

k-rtl و تنظیمات آن در فایل kendo.rtl.min.css قرار دارند که در ابتدای head صفحه تعریف شده است.

## تامین داده و نمایش گرید

در ادامه کدهای کامل DataSource و Kendo UI Grid را ملاحظه می کنید:

```
<script type="text/javascript">
$(function () {
    var productsDataSource = new kendo.data.DataSource({
        transport: {
            read: {
                url: "api/products",
                dataType: "json",
                contentType: 'application/json; charset=utf-8',
                type: 'GET'
            },
            parameterMap: function (options) {
                return kendo.stringify(options);
            }
        },
        schema: {
            data: "Data",
            total: "Total",
            model: {
                fields: {
                    // تعیین نوع فیلد برای جستجوی پویا مهم است
                    "Id": { type: "number" },
                    "Name": { type: "string" },
                    "IsAvailable": { type: "boolean" },
                    "Price": { type: "number" }
                }
            }
        },
        error: function (e) {
            alert(e.errorThrown);
        },
        pageSize: 10,
        sort: { field: "Id", dir: "desc" },
        serverPaging: true,
        serverFiltering: true,
        serverSorting: true
    });

    $("#report-grid").kendoGrid({
        dataSource: productsDataSource,
        autoBind: true,
        scrollable: false,
        pageable: true,
        sortable: true,
        filterable: true,
        reorderable: true,
        columnMenu: true,
        columns: [
            { field: "Id", title: "شماره", width: "130px" },
            { field: "Name", title: "نام محصول",
                template: '<input type="checkbox" #= IsAvailable ? checked="checked" : "" #>'
            },
            { field: "Price", title: "قیمت", format: "{0:c}" }
        ]
    });
});
</script>
```

- با تعاریف مقدماتی Kendo UI DataSource [پیشتر آشنا شده ایم](#) و قسمت read آن جهت دریافت اطلاعات از سمت سرور کاربرد

دارد.

- در اینجا ذکر contentType الزامی است. زیرا ASP.NET Web API بر این اساس است که تصمیم می‌گیرد، خروجی را به صورت JSON ارائه دهد یا XML.
- با استفاده از parameterMap، سبب خواهیم شد تا پارامترهای ارسالی به سرور، با فرمت صحیحی تبدیل به JSON شده و بدون مشکل به سرور ارسال گردند.
- در قسمت schema باید نام فیلدهای موجود در DataSourceResult دقیقاً مشخص شوند تا گرید بداند که data را باید از چه فیلدی استخراج کند و تعداد کل ردیف‌ها در کدام فیلد قرار گرفته‌است.
- نحوه‌ی تعریف model را نیز در اینجا ملاحظه می‌کنید. ذکر نوع فیلدها در اینجا بسیار مهم است و اگر قید نشوند، در حین جستجوی پویا به مشکل برخوردیم خورد. زیرا پیش فرض نوع تمام فیلدها string است و در این حالت نمی‌توان عدد 1 رشته‌ای را با یک فیلد از نوع int در سمت سرور مقایسه کرد.
- در اینجا serverFiltering، serverPaging و serverSorting نیز به true تنظیم شده‌اند. اگر این مقدار دهی‌ها صورت نگیرد، این اعمال در سمت کلاینت انجام خواهند شد.

پس از تعریف DataSource، تنها کافی است آن را به خاصیت dataSource یک kendoGrid نسبت دهیم.

- autoBind: true سبب می‌شود تا اطلاعات DataSource بدون نیاز به فراخوانی متد read آن به صورت خودکار دریافت شوند.
- با تنظیم scrollable: false، اعلام می‌کنیم که قرار است تمام رکوردها در معرض دید قرار گیرند و اسکرول پیدا نکنند.
- pageable: true صفحه بندی را فعال می‌کند. این مورد نیاز به تنظیم pageSize: 10 در قسمت DataSource نیز دارد.
- با sortable: true مرتب سازی ستون‌ها با کلیک بر روی سرستون‌ها فعال می‌گردد.
- filterable: true به معنای فعال شدن جستجوی خودکار بر روی فیلدها است. کتابخانه‌ی Kendo.DynamicLinq حاصل آن را در سمت سرور مدیریت می‌کند.
- reorderable: true سبب می‌شود تا کاربر بتواند محل قرارگیری ستون‌ها را تغییر دهد.
- columnMenu: true اختیاری است. اگر ذکر شود، امکان مخفی سازی انتخابی ستون‌ها نیز مسیر خواهد شد.
- در آخر ستون‌های گرید مشخص شده‌اند. با تعیین format: "{0:c}" سبب نمایش فیلدهای قیمت با سه رقم جدا کننده خواهیم شد. مقدار ریال آن از فایل فرهنگ جاری تنظیم شده دریافت می‌گردد. با استفاده از template تعریف شده نیز سبب نمایش فیلد bool به صورت یک checkbox خواهیم شد.

**کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:**

[KendoUI03.zip](#)

## نظرات خوانندگان

نویسنده: حمیدرضا کبیری  
تاریخ: ۱۹:۳۱ ۱۳۹۳/۰۸/۱۶

آیا kendo UI کاملاً از زبان فارسی پشتیبانی میکند ؟  
برای calender آن ، به تقویم شمی گزینه ای موجود هست ؟  
این [گزینه](#) با ورژن ۲۰۱۴/۱/۳۱۸ مطابقت دارد ، آیا با ورژنهای جدید مشکلی نخواهد داشت ؟

نویسنده: احمد رجبی  
تاریخ: ۲۰:۱۵ ۱۳۹۳/۰۸/۱۶

میتوانید با اضافه کردن [این اسکریپت](#) تمامی قسمتهای kendo را به زبان فارسی ترجمه کنید.

نویسنده: سعیدجلالی  
تاریخ: ۸:۴۴ ۱۳۹۳/۰۸/۱۷

با تشکر از مطلب مفید شما من از wrapper mvc مجموعه kendo استفاده میکنم  
توی مطالب شما در مورد استفاده از [Kendo.DynamicLinq](#) صحبت شد خواستم بدونم آیا وقتی از wrapper هم استفاده میکنیم  
استفاده از این پکیج لازم هست؟

```
@(Html.Kendo().Grid(Model)
    .Name("gridKendo")
    .Columns(columns =>
    {
        columns.Bound(c => c.Name).Width(50);
        columns.Bound(c => c.Family).Width(100);
        columns.Bound(c => c.Tel);
    })
    .Pageable(pager => pager
        .Input(true)
        .Numeric(true)
        .Info(true)
        .PreviousNext(true)
        .Refresh(true)
        .PageSizes(true)
    )
)
```

چون من با استفاده از telerik profiler وقتی درخواست رو بررسی میکنم توی دستور sql چنین دستوری رو در انتها مشاهده میکنم:  
صفحه اول:

```
SELECT *
FROM (
    SELECT
    FROM table a
)
WHERE ROWNUM <= :TAKE
```

صفحات بعد:

```
SELECT *
FROM (
    SELECT
    a.*,
    ROWNUM OA_ROWNUM
    FROM (
    FROM table a
```

```
) a
WHERE ROWNUM <= :TAKE
)
WHERE OA_ROWNUM > :SKIP
```

پایگاه داده اوراکل است.

نویسنده: سعیدجلالی  
تاریخ: ۹:۱۲ ۱۳۹۳/۰۸/۱۷

امکان فارسی شدن تمام بخش‌ها وجود دارد. تقویم هم فارسی شده است در [این سایت](#) برای نسخه‌های جدیدتر هم باید دوتا فایل جاوا اسکریپت all و mvc رو خودتون تغییر بدهید (با توجه به الگوی انجام شده در فایل فارسی شده فوق) ولی برای تقویم زمانبندی scheduler من فارسی ندیده ام

نویسنده: وحید نصیری  
تاریخ: ۹:۳۱ ۱۳۹۳/۰۸/۱۷

مطلب فوق نه وابستگی خاصی به وب فرم‌ها دارد و نه ASP.NET MVC. ویو آن یک فایل HTML ساده است و سمت سرور آن فقط یک کنترلر ASP.NET Web API که با تمام مشتقات ASP.NET سازگار است. در این حالت یک نفر می‌تواند ASP.NET نگارش خودش را خلق کند؛ بدون اینکه نگران جزئیات وب فرم‌ها باشد یا ASP.NET MVC. ضمناً دانش جاوا اسکریپتی آن هم قابل انتقال است؛ چون اساساً Kendo UI برای فناوری سمت سرور خاصی طراحی نشده است و حالت اصل آن با PHP، Java و امثال آن هم کار می‌کند.

نویسنده: میثم آقا احمدی  
تاریخ: ۱۳:۱۷ ۱۳۹۳/۰۸/۱۷

در کنترلر این خط باعث بارگذاری تمامی داده‌ها می‌شود

```
var list = ProductDataSource.LatestProducts;
```

آیا راه حلی وجود دارد که دیتای به تعداد همان pagesize از پایگاه خوانده شود؟

نویسنده: وحید نصیری  
تاریخ: ۱۳:۲۸ ۱۳۹۳/۰۸/۱۷

- این فقط یک مثال هست و منبع داده‌ای صرفاً جهت دموی ساده‌ی برنامه. فقط برای اینکه با یک کلیک بتوانید برنامه را اجرا کنید و نیازی به برپایی و تنظیم بانک اطلاعاتی و امثال آن نداشته باشد.  
- شما در کدها و کوئری‌های مثلاً EF در اصل با یک سری [IQueryable](#) کار می‌کنید. همینجا باید متد الحاقی ToDataSourceResult را اعمال کنید تا نتیجه‌ی نهایی در حداقل بار تعداد رفت و برگشت و با کوئری مناسبی بر اساس پارامترهای دریافتی به صورت خودکار تولید شود. در انتهای کار بجای مثلاً ToList بنویسید ToDataSourceResult.

نویسنده: امین  
تاریخ: ۱۴:۴۱ ۱۳۹۳/۰۸/۱۷

سلام من در ویو خودم نمیتونم اطلاعاتم رو تو kendo.grid بینم و برای من یک لیست استرینگ در ویو نمایش داده میشه و به این شکل در کنترلر و ویو کد نویسی کردم .

```
public class EFController : Controller    {
    //
    // GET: /EF/

    public ActionResult AjaxConnected([DataSourceRequest] DataSourceRequest request )
```



```

{
    using (var dbef=new dbTestEntities())
    {
        IQueryable<Person> persons = dbef.People;
        DataSourceResult result = persons.ToDataSourceResult(request);
        return Json(result.Data,JsonRequestBehavior.AllowGet);
    }
}

```

و ویو

```

@{
    ViewBag.Title = "AjaxConnected";
}

<h2>AjaxConnected</h2>
@(Html.Kendo().Grid<TelerikMvcApp2.Models.Person>(
    .Name("Grid")
    .DataSource(builder => builder
        .Ajax()
        .Read(operationBuilder => operationBuilder.Action("AjaxConnected", "EF"))

    )
    .Columns(factory =>
    {
        factory.Bound(person => person.personId);
        factory.Bound(person => person.Name);
        factory.Bound(person => person.LastName);
    })
    .Pageable()
    .Sortable())

```

و یک لیست استرینگ بهم در عمل خروجی می‌دهد و از خود قالب kendo grid خبری نیست. من اطلاعات رو به طور json پاس میدم و ajax میگیرم.

حالا قبلش همچین خطایی داشتم که به allowget ایراد میگرفت ولی در کل با JsonRequestBehavior.AllowGet حل شد و حالا فقط به لیست بهم خروجی می‌دهد! و از ظاهر گرید خبری نیست. و اگر به جای json نوشته بشه view و با ویو return کنم ظاهر kendo grid رو دارم اما خروجی دارای مقداری نیست! اینم خروجی استرینگ من:

```

[{"personId":1,"Name":"Amin","LastName":"Saadati"}, {"personId":2,"Name":"Fariba","LastName":"Ghochani"}, {"personId":3,"Name":"Rima","LastName":"Rad"}, {"personId":4,"Name":"Milad","LastName":"Rahman"}, {"personId":5,"Name":"Rima","LastName":"Rad"}, {"personId":6,"Name":"Ali","LastName":"Kiva"}, {"personId":7,"Name":"Sahel","LastName":"Abasi"}, {"personId":8,"Name":"Medi","LastName":"Ghaem"}, {"personId":9,"Name":"Mino","LastName":"Kafash"}, {"personId":10,"Name":"Behzad","LastName":"Tizro"}, {"personId":11,"Name":"Toti","LastName":"Saadati"}, {"personId":12,"Name":"Parinaz","LastName":"Karami"}, {"personId":13,"Name":"Sadegh","LastName":"Hojati"}, {"personId":14,"Name":"Milad","LastName":"Ebadipor"}, {"personId":15,"Name":"Farid","LastName":"Riazi"}, {"personId":16,"Name":"Said","LastName":"Abdoli"}, {"personId":17,"Name":"Behzad","LastName":"Ariafootahi"}, {"personId":18,"Name":"Jamshid","LastName":"K"}]

```

این سوال رو در چند سایت پرسیدم و به جوابی برایش نرسیدم. و نمیدونم ایراد کدهای نوشته شده ام کجاست! متشکرم

نویسنده: وحید نصیری  
تاریخ: ۱۳۹۳/۰۸/۱۷ ۱۵:۲

- قصد پشتیبانی از wrapperهای آنرا ندارم. لطفا خارج از موضوع سؤال نپرسید. اگر کسی دوست داشت در این زمینه مطلب منتشر کند، خوب. ولی من چنین قصدی ندارم.

- عرض کردم اگر از wrapperها استفاده کنید، به علت عدم درک زیر ساخت اصلی Kendo UI، قادر به دیباگ کار نخواهید بود.

- اگر متن را مطالعه کنید در قسمت «پیشنیاز تامین داده مخصوص Kendo UI Grid» دقیقا شکل نهایی خروجی JSON مورد نیاز ارائه شده است. این خروجی در سه فیلد data, total و aggregate قرار می‌گیرد. شما الان فقط قسمت data آنرا بازگشت

داده‌اید؛ بجای اصل و کل آن. نام این سه فیلد هم مهم نیست؛ اما هر چیزی که تعیین می‌شوند، باید در قسمت data source در خاصیت schema آن مانند مثالی که در مطلب جاری آمده (در قسمت «تامین داده و نمایش گرید»)، دقیقاً مشخص شوند، تا Kendo UI بداند که اطلاعات مختلف را باید از چه فیلدهایی از JSON خروجی دریافت کند.

نویسنده: وحید محمدطاهری

تاریخ: ۱۴:۲۴ ۱۳۹۳/۱۰/۰۷

با سلام و خدا قوت

آقای نصیری، model ای که باید در قسمت schema تعریف بشه چطوری میشه اونو دینامیک تولید کرد. من یک چنین حالتی رو ایجاد کردم ولی نمی‌دونم چطوری باید اسم ستونو براش مشخص کنم.

```
public class Field
{
    public string Type { get; set; }
}
```

```
public class Fields : System.Collections.ObjectModel.Collection<Field>
{
    public System.Collections.IEnumerable ToList()
    {
        return System.Linq.Enumerable.ToList( System.Linq.Enumerable.Select( this ,
                                                                                   field => new {
field.Type } ) );
    }
}
```

این قسمت اطلاعاتی است که برای ایجاد گرید باز گردانده می‌شود.

```
return new InitializeInfo
{
    ...
    Model = GetModel()
};
```

```
private Model GetModel ()
{
    return new Model{ Fields = GetFields().ToList() };
}
```

متد GetColumns شامل 3 ستون می‌باشد که نوع، عنوان و سایر مشخصات رو توش تعریف کردم

```
private Fields GetFields()
{
    var fields = new Fields();
    foreach ( var column in GetColumns() )
    {
        fields.Add( new Field { Type = column.DataType } );
    }
    return fields;
}
```

الان خروجی که تولید میشه اینجوریه

```
"model": {
  "fields": [
    {
      "type": "string"
```

```

    },
    {
      "type": "string"
    },
    {
      "type": "datetime"
    }
  ]
}

```

ممنون میشم به راهنمایی کنید.

نویسنده: وحید نصیری  
تاریخ: ۱۳۹۳/۱۰/۰۷ ۱۴:۴۱

- پویا هست و خروجی دسترسی هم گرفتید. زمانیکه تعریف می کنید:

```
new Field { Type = column.DataType }
```

یعنی در لیست نهایی، خاصیتی با نام ثابت Type و با مقدار متغیر column.DataType را تولید کن (نام خاصیت، مقدار ثابت نام خاصیت را در JSON نهایی تشکیل می دهد).  
+ نیازی هم به این همه پیچیدگی نداشت. تمام کارهایی را که انجام دادید با تهیه خروجی ساده List<Field> از یک متد دلخواه، یکی هست و نیازی به anonymous type کار کردن نبود.  
- به همان کلاس فیلد، خواص دیگر مورد نیاز را اضافه کنید (عنوان و سایر مشخصات یک فیلد) و در نهایت لیست ساده List<Field> را بازگشت دهید. هر خاصیت کلاس Field، یک ستون گرید را تشکیل می دهد.  
- همچنین دقت داشته باشید اگر از روش مطلب جاری استفاده می کنید، اطلاعات ستون های نهایی باید در فیلد Data نهایی قرار گیرند (قسمت «پیشنیاز تامین داده مخصوص Kendo UI Grid» در بحث).

نویسنده: وحید محمدطاهری  
تاریخ: ۱۳۹۳/۱۰/۰۷ ۱۵:۴۸

با تشکر از پاسختون

درسته این به صورت پویا تولید میشه ولی شکل model ای که شما در این مطلب توضیح دادید با این چیزی که کد من تولید می کنه فرق میکنه

برای شما اول نام فیلد هست بعد نوع اون فیلد، در حالی که نحوه تولید داینامیک اینو نمی دونم چطوری باید باشه.

```

model: {
  fields: {
    "Id": { type: "number" }, // تعیین نوع فیلد برای جستجوی پویا مهم است
    "Name": { type: "string" },
    "IsAvailable": { type: "boolean" },
    "Price": { type: "number" }
  }
}

```

نویسنده: وحید نصیری  
تاریخ: ۱۳۹۳/۱۰/۰۷ ۱۶:۱۹

باید از Dictionary استفاده کنید برای تعریف خواص پویا:

```

public class Field
{
    [JsonExtensionData]
    public Dictionary<string, object> Property { get; set; }
}

```

```
}  
  
public class FieldType  
{  
    public string Type { get; set; }  
}
```

و بعد نحوه استفاده از آن به صورت زیر خواهد بود:

```
var data = new  
{  
    model = new  
    {  
        fields = new List<Field>  
        {  
            new Field  
            {  
                Property = new Dictionary<string, object>  
                {  
                    {"Id", new FieldType { Type = "number" }},  
                    {"Name", new FieldType { Type = "string" }}  
                }  
            }  
        }  
    }  
};  
var dataJson = JsonConvert.SerializeObject(data, Formatting.Indented);
```

با این خروجی:

```
{  
  "model": {  
    "fields": [  
      {  
        "Id": {  
          "Type": "number"  
        },  
        "Name": {  
          "Type": "string"  
        }  
      }  
    ]  
  }  
}
```

- اگر از Web API استفاده می‌کنید، ذکر سطر `JsonConvert.SerializeObject` ضروری نیست و به صورت توکار از [JSON.NET](https://www.json.net/) استفاده می‌کند.

- اگر از ASP.NET MVC استفاده می‌کنید، نیاز است [از آن کمک بگیرید](#). از این جهت که خاصیت `JsonExtensionData` سبب می‌شود تا نام ثابت خاصیت `Property`، از خروجی نهایی حذف شود و اعضای دیکشنری، جزئی از خاصیت‌های موجود شوند.

- نکته‌ی «[گرفتن خروجی CamelCase از JSON.NET](#)» را هم باید مد نظر داشته باشید.