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);
                    }
                }
            }
        }
```