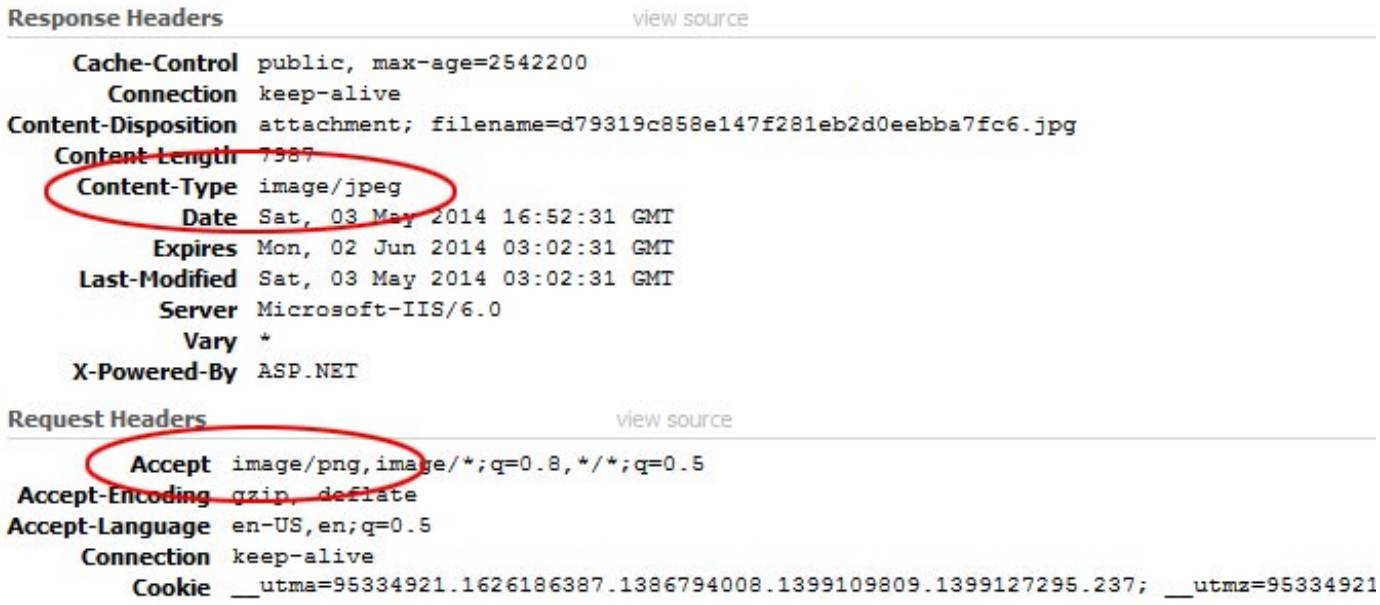
عنوان: **پیاده سازی یک MediaTypeFormatter برای پشتیبانی از MultiPart/form-data در Web API**
نویسنده: محسن درپرستی
تاریخ: ۱۳۹۳/۰۲/۱۴ ۰:۳۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API, MediaTypeFormatter, Multipart-form-data

Media Type یا MIME Type نشان دهنده فرمت یک مجموعه داده است. در HTTP، مدیا تایپ بیان کننده فرمت message body یک درخواست / پاسخ است و به دریافت کننده اعلام می‌کند که چطور باید پیام را بخواند. محل استاندارد تعیین Mime Type در هدر Content-Type است. درخواست کننده می‌تواند با استفاده از هدر Accept لیستی از MimeType های قابل قبول را به عنوان پاسخ، به سرور اعلام کند.



Asp.net Web API از MimeType برای تعیین نحوه serialize یا deserialize کردن محتوای دریافتی / ارسالی استفاده می‌کند

Web API **MediaTypeFormatter** برای خواندن/درج پیام در بدنه درخواست/پاسخ از MediaTypeFormmater ها استفاده می‌کند. اینها کلاس‌هایی هستند که نحوه‌ی Serialize کردن و deserialize کردن اطلاعات به فرمت‌های خاص را تعیین می‌کنند. Web API به صورت توکار دارای formatter هایی برای نوع‌های BSON ، JSON ، XML و Form-UrlEncoded می‌باشد. همه این‌ها کلاس پایه MediaTypeFormatter را پیاده سازی می‌کنند.

مسئله

یک پروژه Web API بسازید و view model زیر را در آن تعریف کنید:

```
public class NewProduct
    {
        [Required]
        public string Name { get; set; }

        public double Price { get; set; }

        public byte[] Pic { get; set; }
    }
```

همانطور که می‌بینید یک فیلد از نوع byte[] برای تصویر محصول در نظر گرفته شده است.
حالا یک کنترلر API ساخته و اکشنی برای دریافت اطلاعات محصول جدید از کاربر می‌نویسیم :

```
public class ProductsController : ApiController
    {
        [HttpPost]
        public HttpResponseMessage PostProduct(NewProduct model)
        {
            if (ModelState.IsValid)
            {
                // ثبت محصول

                return new HttpResponseMessage(HttpStatusCode.Created);
            }

            return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
        }

    }
```

و یک صفحه html به نام index.html که حاوی یک فرم برای ارسال اطلاعات باشد :

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1>ساخت MediaTypeFormatter برای Multipart/form-data</h1>
    <h2>محصول جدید</h2>

    <form id="newProduct" method="post" action="/api/products" enctype="multipart/form-data">
        <div>
            <label for="name">نام محصول : </label>
            <input type="text" id="name" name="name" />
        </div>
        <div>
            <label for="price">قیمت : </label>
            <input type="number" id="price"  name="price" />
        </div>
        <div>
            <label for="pic">تصویر : </label>
            <input type="file" id="pic" name="pic" />
        </div>
        <div>
            <button type="submit">ثبت</button>
        </div>
    </form>
</body>
</html>
```

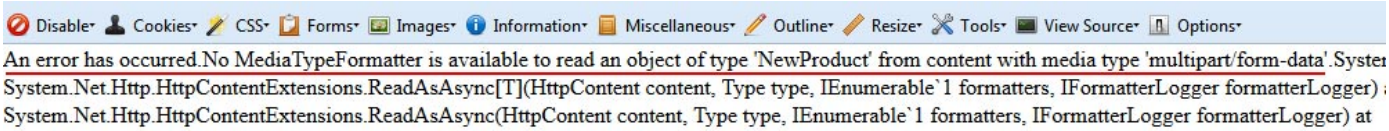
زمانی که فرم حاوی فایلی برای آپلود باشد مشخصه encType باید برابر با Multipart/form-data مقداردهی شود تا اطلاعات فایل به درستی کد شوند. در زمان ارسال فرم Content-type درخواست برابر با Multipart/form-data و فرمت اطلاعات درخواست ارسالی به شکل زیر خواهد بود :

همانطور که می‌بینید هر فیلد در فرم، در یک بخش جداگانه قرار گرفته است که با خط چین هایی از هم جدا شده اند. هر بخش، header های جداگانه خود را دارد.

- Content-Disposition که نام فیلد و نام فایل را شامل می‌شود .

- content-type که mime type مخصوص آن بخش از داده‌ها را مشخص می‌کند.

پس از اینکه فرم را تکمیل کرده و ارسال کنید ، با پیام خطای زیر مواجه می‌شوید :



خطای روی داده اعلام می‌کند که Web API فاقد MediaTypeFormatter برای خواندن اطلاعات ارسال شده با فرمت MultiPart/Form-data است. Web API برای خواندن و بایند کردن پارامترهای complex Type از درون بدنه پیام یک درخواست از MediaTypeFormatter استفاده می‌کند و همانطور که گفته شد Web API فاقد Formatter توکار برای deserialize کردن داده‌های با فرمت Multipart/form-data است.

راه حل‌ها :

روشی که در سایت asp.net برای آپلود فایل در web api استفاده شده، عدم استفاده از پارامترها و خواندن محتوای Request در درون کنترلر است. که به طبع در صورتی که بخواهیم کنترلرهای تمیز و کوچکی داشته باشیم روش مناسبی نیست. از طرفی امتیاز parameter binding و modelstate را هم از دست خواهیم داد.

روش دیگری که می‌خواهیم در اینجا پیاده سازی کنیم ساختن یک MediaTypeFormatter برای خواندن فرمت Multipart/form-data است. با این روش کد موردنیاز کپسوله شده و امکان استفاده از binding و modelstate را خواهیم داشت.

برای ساختن یک MediaTypeFormatter یکی از 2 کلاس MediaTypeFormatter یا BufferedMediaTypeFormatter را باید پیاده سازی کنیم . تفاوت این دو در این است که BufferedMediaTypeFormatter برخلاف MediaTypeFormatter از متدهای synchronous استفاده می‌کند.

پیاده سازی :

یک کلاس به نام MultiPartMediaTypeFormatter می‌سازیم و کلاس MediaTypeFormatter را به عنوان کلاس پایه آن قرار می‌دهیم .

```
public class MultiPartMediaTypeFormatter : MediaTypeFormatter
{
    ...
}
```

ابتدا در تابع سازنده کلاس فرمت هایی که می‌خواهیم توسط این کلاس خوانده شوند را تعریف می‌کنیم :

```
public MultiPartMediaTypeFormatter()
        {
            SupportedMediaTypes.Add(new MediaTypeHeaderValue("multipart/form-data"));
        }
```

در اینجا Multipart/form-data را به عنوان تنها نوع مجاز تعریف کرده ایم.

سپس با پیاده سازی توابع CanReadType و CanWriteType مربوط به کلاس MediaTypeFormatter مشخص می‌کنیم که چه مدل‌هایی را می‌توان توسط این کلاس serialize / deserialize کرد. در اینجا چون می‌خواهیم این کلاس محدود به یک مدل خاص نباشد، از یک اینترفیس برای شناسایی کلاس‌های مجاز استفاده می‌کنیم .

```
public interface INeedMultiPartMediaTypeFormatter
    {
    }
```

و آنرا به کلاس newProduct اضافه می‌کنیم :

```
public class NewProduct : INeedMultiPartMediaTypeFormatter
{
 ...
}
```

از آنجا که تنها نیاز به خواندن اطلاعات داریم و قصد نوشتن نداریم، در متد CanWriteType مقدار false را برمی گردانیم .

```
public override bool CanReadType(Type type)
        {
            return typeof(INeedMultiPartMediaTypeFormatter).IsAssignableFrom(type);
        }

        public override bool CanWriteType(Type type)
        {
            return false;
        }
```

و اما تابع ReadFromStreamAsync که کار خواندن محتوای ارسال شده و بایند کردن آنها به پارامترها را برعهده دارد

```
public async override Task<object> ReadFromStreamAsync(Type type, Stream stream, HttpContent content,
IFormatterLogger formatterLogger)
```

که در آن پارامتر type مربوط به مدل مشخص شده به عنوان پارامتر اکشن (NewProduct)است و پارامتر content محتوای

درخواست را در خود دارد.

ابتدا محتوای ارسال شده را خوانده و اطلاعات فرم را استخراج می‌کنیم و از طرف دیگر با استفاده از کلاس Activator یک نمونه از مدل جاری را ساخته و لیست property های آنرا استخراج می‌کنیم.

```
MultipartMemoryStreamProvider provider = await content.ReadAsMultipartAsync();
            IEnumerable<HttpContent> formData = provider.Contents.AsEnumerable();

            var modelInstance = Activator.CreateInstance(type);
            IEnumerable<PropertyInfo> properties = type.GetProperties();
```

سپس در یک حلقه به ترتیب برای هر property متعلق به مدل، در میان اطلاعات فرم جستجو می‌کنیم. برای پیدا کردن اطلاعات متناظر با هر property در هدر Content-Disposition که در بالا توضیح داده شد، به دنبال فیلد همنام با property می‌گردیم .

```
foreach (PropertyInfo prop in properties)
            {
                var propName = prop.Name.ToLower();
                var propType = prop.PropertyType;

                var data = formData.FirstOrDefault(d =>
d.Headers.ContentDisposition.Name.ToLower().Contains(propName));
```

در صورتی که فیلدی وجود داشته باشد کار را ادامه می‌دهیم .

گفتیم که هر فیلد یک هدر، Content-Type هم می‌تواند داشته باشد. این هدر به صورت پیش فرض معادل text/plain است و برای فیلدهای عادی قرار داده نمی‌شود . در این مثال چون فقط یک فیلد غیر رشته ای داریم فرض را بر این گرفته ایم که در صورت وجود Content-Type ، فیلد مربوط به تصویر است. در صورتیکه ContentType وجود داشته باشد، محتوای فیلد را به شکل Stream خوانده به byte[] تبدیل و با استفاده از متد SetValue در property مربوطه قرار می‌دهیم.

```
if (data != null)
                {
                    if (data.Headers.ContentType != null)
                    {
                        using (var fileStream = await data.ReadAsStreamAsync())
                        {
                            using (MemoryStream ms = new MemoryStream())
                            {
                                fileStream.CopyTo(ms);
                                prop.SetValue(modelInstance, ms.ToArray());
                            }
                        }
                    }
```

در صورتی که Content-Type غایب باشد بدین معنی است که محتوای فیلد از نوع رشته است (عدد ، تاریخ ، guid ، رشته) و باید به نوع مناسب تبدیل شود. ابتدا آن را به صورت یک رشته می‌خوانیم و با استفاده از Convert.ChangeType آنرا به نوع مناسب تبدیل می‌کنیم و در property متناظر قرار می‌دهیم .

```
if (data != null)
                {
                    if (data.Headers.ContentType != null)
                    {
                        //...
                    }
                    else
                    {
                        string rawVal = await data.ReadAsStringAsync();
                        object val = Convert.ChangeType(rawVal, propType);

                        prop.SetValue(modelInstance, val);
                    }
                }
```

و در نهایت نمونه ساخته شده از مدل را برگشت می‌دهیم.

```
return modelInstance;
```

برای فعال کردن این Formatter باید آنرا به لیست formmaterهای web api اضافه بکنیم. فایل WebApiConfig در App_Start را باز کرده و خط زیر را به آن اضافه می‌کنیم:

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

حال اگر مجددا فرم را به سرور ارسال کنیم، با پیام خطایی، مواجه نشده و عمل binding با موفقیت انجام می‌گیرد.

```
public class ProductsController : ApiController
{
    [HttpPost]
    public HttpResponseMessage PostProduct(NewProduct model)
    {
        if (ModelState.IsValid)
        {
            //  implemenet rest of the business

            return new HttpResponseMessage(HttpStatusCode.Created);
        }

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }
```